intel

8086
SOFTWARE TOOLBOX

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

Order Number: 122203-002

8086
SOFTWARE TOOLBOX

Order Number: 122203-002

Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA 95051

Additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

Intel retains the right to make changes to these specifications at any time, without notice. Contact your local sales office to obtain the latest specifications before placing your order.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

The following are trademarks of Intel Corporation and its affiliates and may only be used to identify Intel products:

BITBUS	i_m	iSBC	Plug-A-Bubble
COMMputer	iMMX	iSBX	PROMPT
CREDIT	Insite	iSDM	Promware
Data Pipeline	int_el	iSXM	QueX
Genius	int_elBOS	KEPROM	QUEST
i	Intelevision	Library Manager	Ripplemode
i^2	$int_eligent$ Identifier	MCS	RMX/80
I^2ICE	$int_eligent$ Programming	Megachassis	RUPI
ICE	Intellec	MICROMAINFRAME	Seamless
iCS	Intellink	MULTIBUS	SOLO
iDBP	iOSP	MULTICHANNEL	SYSTEM 2000
iDIS	iPDS	MULTIMODULE	UPI
iLBX	iRMX		

REV.	REVISION HISTORY	DATE	APPD.
-001	Original issue.	12/83	J.P.
-002	Revision 1.1.	1/85	J.P.

```
++=============================================++
||                                             ||
||            8086 SOFTWARE TOOLBOX            ||
||                 VERSION 1.1                 ||
||                   PREFACE                   ||
||                                             ||
++=============================================++
```

This manual defines and describes the use of the 8086 Software Toolbox. The Toolbox is a collection of utilities which can help improve programmer productivity.

Listed below is a brief description of each tool for quick reference.

Tools marked with a "*" can utilize extra 8086 system memory, meaning they can execute more efficiently, process more symbols, and/or accept a larger variety of input.

Text Formatting
- * SCRIPT batch text formatter
- MPL standalone macro processor
- * SPELL spelling checker program, plus dictionaries
- * WSORT SPELL dictionary compactor

80286 Tools
- * OMC286 8086 to 80286 object module converter
- E80287 80286-based 80287 emulator, plus initialization library E80287.LIB

Performance Analysis
- * PERF code performance analysis tool
- GRAFIT PERF map grapher

Sorting
- * ESORT very flexible sort program
- * HSORT in memory heap sort program

Miscellaneous
- * XREF list file inter-module cross reference tool plus XREF80 for asm80,asm48 listings
- DC floating-point calculator plus DC87, uses 8087 hardware
- FUNC SERIES IIIE function key programmer
- PASSIF general purpose assertion checking and reporting tool
- * COMP synchronized file compare program

```
++=============================================++
||                                             ||
||            8086 SOFTWARE TOOLBOX            ||
||                 VERSION 1.1                 ||
||                  GLOSSARY                   ||
||                                             ||
++=============================================++
```

UDI: refers to the standard INTEL 8086/8088 based operating system service routine interface.

For details, see the following manuals:

- Intellec Series-IV Operating and Programming Guide 121753
- Intellec Series III Microcomputer Development System Programmer's Reference Manual 121618
- iRMX 86 Programmer's Reference Manual, Part II 146196

8086 SOFTWARE TOOLBOX
VERSION 1.1
TABLE OF CONTENTS

```
+==================================+
|                                  |
|             SCRIPT               |
|              v1.1                |
|                                  |
+==================================+
```

I. INTRODUCTION

The iMDX Text Formatter (SCRIPT) formats an input text file under control of commands embedded in the text. SCRIPT is useable on any UDI-compatible 8086-based system with at least 96K of user memory. Output is to the DTC-KRS, DTC-300, any line printer or UDI-supported file.

SCRIPT uses the file handling capabilities provided by UDI. Input files are prepared using a separate text editor such as AEDIT, CREDIT, etc. (SCRIPT will concatenate input files, permitting the use of memory resident editors, even on large documents.) Uppercase-only files are fully handled by SCRIPT, as are upper/lower case files.

Not all of the features described have been implemented in the current version of SCRIPT, but they are being specified to make sure the current design does not preclude their later implementation. Those features that are not implemented in the current version are hashed out in this document.

II. INTERFACE SPECIFICATIONS

A. User Interface

1. Characters

Certain characters in the input text cause special actions to take place. For example, a space delimits words, and ampersands surround text to be underlined. Many of these special functions can be assigned to a character of the users choice. In addition, a literal character (initially ^) causes the the character that follows it to be treated as a plain text character.

As far as the user is concerned, line boundaries in the input text are ignored. To make this happen, SCRIPT must take special action at the end of an input line. In most cases, it must add a space to the end of the last word on the line so that it doesn't run into the first word of the

next line. If the word looks like the end of a sentence, SCRIPT will add two spaces. If the word ends with a hyphen, however, it is assumed to be a hyphenated phrase, and no extra blanks are inserted. But, if the "word" is actually a dash made of two hyphens, a space is inserted.

In fill mode (see below), SCRIPT will add spaces after any word that looks like the end of a sentence to ensure that it is followed by at least two spaces.

Many text formatters remove "extra" spaces found in the input file. Since it is desireable to have two spaces at the end of every sentence, these formatters must try to recognize the end of a sentence to do so. Also, they must provide a mechanism for inserting extra spaces when they really are needed (for special symbols to be added later, for instance). SCRIPT will never be as clever as its users, so if a user puts in spaces, SCRIPT will leave them there. The only spaces that SCRIPT will delete are those marking the beginning of a paragraph (see the AUTOPARAGRAPH command).

2. Commands

SCRIPT commands are block-structured. BEGIN and END commands bracket blocks, and internal variables such as margins and paragraph spacing may be given new values within the block. These new values hold from the point of assignment to the end of the smallest enclosing block, or until they are again reassigned. All such variables initially have default values.

A command may appear anywhere in an input line. The start and end of a command group are signalled by a special character (initially '/'). Multiple commands are separated by semicolons or linefeeds, and arguments are separated by commas. (Any of these characters may be used in command arguments by using the literal character in front of them.) A null command is tolerated so that other commands may be terminated by a semicolon. For most commands, arguments may be omitted and default values will be used. SCRIPT looks only at the first letter and the remaining consonants of a command, and accepts as a command any string of consonants that is the valid prefix of one of the commands listed below. Selected commands have alternative abbreviations in addition to this abbreviation rule. Commands may be written in any combination of upper and lower case. The commands (and their nonstandard abbreviations) are:

ASIS - - - END

The ASIS ("as is") command causes a break and begins a

block, in which SCRIPT processes text in the NOFILL, NOJUSTIFY, NOAUTOPARAGRAPH mode. In this mode plain text will appear in the output just as it does in the input. (All other commands, such as SPACING, UPPERCASE, and LMARGIN, are active, however. Macros are expanded.)

AUTOPARAGRAPH (AP) [-] i,s,t

Sets a mode in which a null line or any line starting with a space is interpreted as the start of a new paragraph, i.e. as if it had been preceded by a PARAGRAPH command(this will cause a BREAK). The leading spaces(s) are removed from the input line. If any parameters are given, they redefine the defaults for the PARAGRAPH command in the current block.

NOAUTOPARAGRAPH (NAP)

Lines with leading spaces are interpreted just like other lines. The leading spaces are not removed.

AUXFILE [n] {text}

The "text" delimited by "{" and "}" is written to auxiliary file n. The default file is auxiliary file Ø. (The mapping from auxiliary file numbers to physical files is established by an invocation control as described in Section III.) The text may be any string of characters that is balanced with respect to "{"-"}" bracketing. (Unbalanced braces may be introduced using the "literalizing" character.) The text is not processed the same as text going to the main output file. Only macro calls within the "text" are expanded before writing to the file. All other SCRIPT commands and special characters are NOT processed.

BEGIN [name] - - - END [name]

BEGIN starts a block. The name is used for error checking with the name on the end command. END marks the end of a block. All parameters set within the block revert to their previous values. The parameters that could be affected are LMARGIN; RMARGIN; bar margin; AUTOPARAGRAPH; PARAGRAPH indent, skip, and test values; CENTER; FILL; JUSTIFY; LOWERCASE/UPPERCASE/NOCASE; SPACING; TAB settings; FLAG characters; drop character; bar character; and USPACE. The name on the END command should match that on the corresponding BEGIN command. It is possible to detect that an END command has been left out, and SCRIPT could try to correct the situation. (Names on blocks are not implemented in this version.)

Neither a BEGIN nor an END command cause a BREAK.

BREAK [n]

Ends the current output line and leaves n blank lines (regardless of the current SPACING mode). The default value of n is 0. Lines ended by explicit or implicit BREAKs are never justified. In an attempt to prevent "widows" (printing the last line of a paragraph alone at the top of a page), SCRIPT will allow a line ended by a BREAK to occupy the line after the "last" text line in a page.

CENTER

This command causes a BREAK and switches SCRIPT to the centering mode. Output lines are centered between left and right margins. Centered lines are not filled or justified. Lines with tabs in them are not centered.

NOCENTER

This command causes a break and turns off centering of output lines. Note: CENTER must be in effect at the END of an output line to cause the line to be centered. Thus, /begin; center/ ... /end/ will not generate a centered line because CENTER will be turned off by the /end/, before the end of the line. Because /nocenter/ causes a break before turning off centering, all lines between /center/ ... /nocenter/ are sure to be centered.

~~COLUMN [n]~~

~~This command causes a break and switches output to column n. (The default column is the next defined column modulo the number of columns.) Output for each column is sent to a separate buffer. Then, when an implicit or explicit COLUMN command switches output to a column that is the same as or is to the left of the current column, all saved text is output together. Short columns are padded with blank lines to match the longest column. Columns may not be longer than a page. Outside a table, COLUMN n is treated as a PAGE command. Use the TABLE command to set column dimensions.~~

COMMENT text

This command is ignored. The comment text may not contain a semicolon.

DROP [n] [,c]

Tabs to the n-th tab stop, "dropping" the character c as it moves. If n is not coded or is Ø, the next tab stop to the right is selected. If the selected tab stop is to the left of the current position, a BREAK is executed, and the new line is started at the tab position. If the current position is the selected tab position, no movement occurs. Drop characters are not generated to the left of the start of the line. c becomes the default drop character until the next /end/ or /drop,c/ command is executed. The drop character is initially ".".

Example: /settabs +10,+20,+30,+40,+50; break/
/drop 2,-/+/drop/+/drop5/+
/drop4,*/
generates:

```
--------------------+---------+--------------------+
****************************************
```

/ FIGURE n,m/- + -/END / / / / / / / / / / / / / /
/ /
/ / This/command causes/a BREAK./ If enough space remains on
/ / the/current page to skip/n /lines/ and /print/ the /text /
/ /generated by the input between the paired/FIGURE and END/
/ / commands, SCRIPT/does so./ Otherwise,/ the /figure text
/ / and/ caption/ are /saved/ for /the beginning of the/next /
/ /page,/and normal text may continue on the/current /page./
/ / After the/ figure /is/ placed /on/ a/ page/ BREAK n is
/ / executed./ The figure/text is implicitly enclosed in /a /
/ /begin-end block/and may/not contain/a KEEP, FIGURE/or/
/ / PAGE/command./ (At /this time, FIGURE/m,n -/- + END/ is
/ / equivalent to PAGE/ BEGIN -/- + END;/ BREAK m/) / / / /

FILL

Sets FILL mode (the initial condition), which causes words from separate input lines to run together to make output lines as long as possible without overflowing the left and right margins.

NOFILL

This command causes a BREAK and clears FILL mode. The end of every input text line causes the end of an output line. Blank input lines cause blank output lines. As in the FILL mode, leading blanks on a line or a null line will initiate an AUTOPARAGRAPH if that mode is active, and if JUSTIFY is set, the line will be

justified unless it is followed by a null line . Under NOFILL, if an input line is too long for the output margins, the RIGHT MARGIN IS IGNORED and text is allowed to extend indefinitely to the right. Note that lines containing only commands are not treated specially in NOFILL mode--they will generate blank output lines.

FLAG function [,character]

Assigns a special function to the given character. This assignment holds until it is reassigned, or until the end of the block. Whatever character previously held the special function loses its special function. If no character is given, this command does nothing but "unspecial" the old character. Any other special function held by the given character is also lost. Legal flag functions, together with the flag's effect, are:

ASCII — Marks the beginning of a decimal number that is interpreted as the ASCII code for a single character.

BACKSPACE — Generates a backspace code in the output.

CONTROL — The next character is treated as an "invisible" control character. The EXTEND or LITERAL or ASCII flag characters may be used to enter the control character.

COMMAND — Marks the beginning and end of a command string. Initially '/'.

DELIMITER — Separates arguments in a macro call. The default value is "\".

DIVIDE — Marks where a word can be divided. If it helps the output line look better, SCRIPT divides the word where this character appears (and inserts a hyphen). Otherwise the character is discarded. For example:

```
/begin; fill;just; lm40;rm60; FLAG DIVIDE,`/
Anti`dis`estab`lish`mentar`ianism is
super`cali`fragil`istic`expi`alidou`sious!?
/break; end/
```

generates

```
Antidisestablish-
mentarianism      is
supercalifragil-
```

isticexpialidou-
sious!?

EXTEND — Adds the parity bit to the following character, producing an extended character set. All extended characters are assumed to be one space wide.

LITERAL — Treats the next character as a simple text character without changing it or interpreting it. Initially '^'.

MACRO — Marks the beginning of a macro call, macro formal parameter, or macro definition. Initially "%".

NTB — Stands for a Non-Trivial Blank. The non-trivial blank prints as a blank, but is treated as an ordinary text character, otherwise. Initially a tilda (i.e., 7EH).

SUBSCRIPT — Marks the beginning of a subscript expression. On the DTC-300 the characters that follow the flag character are printed 3/48-th inch lower than the preceding characters. On other devices, the characters are lowered a full line. Subscripting is ended by a SUPERSCRIPT character. Subscripts may be subscripted to a depth of 64.

SUPERSCRIPT — Marks the beginning of a superscript expression. On the DTC-300 the following characters are printed 3/48-th inch higher than the preceding characters. On other devices, the characters are raised a full line. Superscripting is ended by a SUBSCRIPT character. Superscripts may be superscripted to a height of 64.

Example:

```
"/FLAG BCKSPC,`; FLAG SBSCR,{; FLAG SPRSCR,}; lm+7;rm-7/ Each a{i} has a cooling factor equivalent to e}(x-y}3{){, which means that each M}2{`{a{i},b{j}} is only marginally stable at 400}o{F. On the other hand, a larger coefficient of cooling, with the appropriate scale factors applied, would not necessarily mean stability would return at 450}o{F."
```

would produce

> Each a_i has a cooling factor equivalent to $e^{(x-y^3)}$, which means that each M_{a_i,b_j}^{2} is only marginally stable at 400 ^{o}F. On the other hand, a larger coefficient of cooling, with the appropriate scale factors applied, would not necessarily mean stability would return at 450 ^{o}F.

UNDERLINE Marks the beginning and end of text that is to be underlined. Spaces not in the margins but between underlined words are underlined according to the setting of the USPACE and NUSPACE commands. Initially '&'.

FOOT/EFOOT/OFOOT [left] [,[middle] [,right]]

Specify the footers to be placed on each page. EFOOT specifies the footer for even-numbered pages, OFOOT specifies the footer for odd-numbered pages, and FOOT specifies the same footer for both even- and odd-numbered pages. The left string is justified to the left margin in effect at the time of the command, the middle string is centered between the margins, and the right string is justified on the right margin. If the center string consists of one non-blank character (other than '#'), that character is replicated throughout the central field (from the end of the right string to the beginning of the left string). When the footer is printed, a number sign (#) is replaced by the current page number. The foot is initially empty.

HASH [n] [,char] - - - ENDHASH [n] [,char]

The HASH-ENDHASH commands delimit a region of text which will be overstruck with "char"s when printed. The region will be overstruck only if the current hash level (set by the HASHLEVEL command) is n or less. Hash regions to be printed may not be nested. The initial "char" value is "/". The hash character value is not

affected by the nesting of BEGIN-END blocks. HASH blocks are used in this document to "shade" the features that are not supported in the present release of SCRIPT.

HASHLEVEL [n] [,Hmar] [,Hoff]

HASHLEVEL sets the current hash level and sets two parameters that control the overstriking operation. Within a HASH-ENDHASH region, the overstriking "char"s are printed between columns "Hmar" and width-"Hmar" separated by "Hoff" spaces. From line to line, the first column where the hash "char" is printed cycles from "Hmar" to "Hmar"+"Hoff"-1. The initial settings are equivalent to a HASHLEVEL Ø,2,3 command. HASH blocks in this document use the initial settings for Hmar and Hoff.

HEAD/EHEAD/OHEAD [left] [,[middle] [,right]]

Similar to FOOT commands, but specifying page headings for each page, or each even-, or odd- numbered page. The initial heading is /head,,PAGE #/. To turn off a head or foot, issue the appropriate command with no parameters. The end of a block does not cause a head or foot to revert to its previous value.

INDENT [-] i

This command causes a BREAK. The next line of output is "indented" i spaces from the left margin. If a minus sign precedes "i", the line is started i spaces to the left of the left margin, if possible.

ITAB [n]

Tabs to the n-th tab stop and establishes the left margin there, until the next implicit or explicit BREAK command. If n is not coded or is Ø, then the next tab stop to the right is selected. If the selected tab stop is to the left of the current position, a BREAK is executed, and the new line is started at the tab position. If the current position is the selected tab position, no movement occurs. When the BREAK occurs, the left margin will be set to whatever it would have been, had no ITAB command been present. ITAB is similar to the MTAB command, except that the left margin set by ITAB is only in effect until the next BREAK.

ITAB is useful in formatting lists of short definitions. For example, the following input could be used to generate the initial flag definitions given above:

/begin; ap 0,1,0; settabs +12/

ASCII/itabl/Marks the beginning of a /erv3/decimal/erv3/ number that is interpreted as the ASCII code for a single character. /rev3;rev3/

COMMAND/itabl/Marks the beginning and end of a command string. Initially '^/'.

DIVIDE/itabl/Marks where a word can be divided. If it helps the output line look better, SCRIPT divides the word where this character appears (/rev3/and

JUSTIFY

Sets JUSTIFY mode, in which every output line not ended by a BREAK is expanded by inserting full or partial spaces (depending on the invocation option) to right-justify the text on the right margin. Blanks are inserted starting from the left or right depending on the parity of the line number. No blanks are inserted to the left of the left margin, to the left of the first word on the line, to the left of a tabulated word, or anywhere in a centered line. JUSTIFY is the initial mode.

NOJUSTIFY

Turns off JUSTIFY mode. No blanks are inserted into the output line, making an uneven right margin possible.

KEEP - - - ENDK

The KEEP command causes a BREAK. If the output text generated between paired KEEP and ENDK commands will fit on the current page, SCRIPT will leave it there. Otherwise, a new page is started and, after any pending figures have been placed on pages, the KEEP text is placed on that page. KEEP may not contain a PAGE, KEEP or FIGURE command. (At this time, KEEP - - - ENDK is equivalent to PAGE; BEGIN - - - END.)

LENGTH n [,[hm] [,[tm] [,[bm] [,fm]]]]

Declares the length of the paper to be n lines and establishes the vertical margins for headers, text and footers. With 1/6 inch spacing, 11 inch paper is 66 lines long and 14 inch paper is 84 lines long. hm specifies the number of blank lines above the header; tm specifies the number of lines above the first line of text (must be greater than or equal to hm); bm specifies

the number of lines after the last text line to the bottom of the page; and fm specifies the number of lines below the footer to the bottom of the page. The initial settings are equivalent to /LENGTH 66,2,6,6,2/, so that the default paper length is 66 lines, the heading starts on line 3, text starts on line 7, the last text line is 7 lines from the end of the paper, and the footer is placed on the third line from the end.

LMARGIN [+ | -] n

This command sets the left margin and causes a BREAK . If a plus or minus sign precedes n, the previous left margin is incremented or decremented by n, respectively. The initial value is zero.

SCRIPT outputs a line at a time to the output file. After writing out a line, it then sets the left margin for the next output line. Once set, the left margin for a line does not change. That is why, for example, the LMARGIN command causes a break: so that the text following the LMARGIN command is indeed placed on a line with the specified left margin.

LOWERCASE

Uppercase letters in the input file are converted to lowercase. The LITERAL flag (initially ^) can override this conversion. This is the initial mode.

MTAB [n]

Tabs to the n-th tab stop and establishes the left margin there. If n is not coded or is Ø, the next tab stop to the right is selected. If the selected tab stop is to the left of the current position, a BREAK is executed, and the new line is started at the tab position. If the current position is the selected tab position, no movement occurs. Also see the ITAB command.

MTAB is useful in formatting lists of short definitions. For example, the following input generated the initial flag definitions given above:

/begin; settabs +Ø,+12/

/skip;m1/ASCII/m2/Marks the beginning of a /erv3/decimal/erv3/ number that is interpreted as the ASCII code for a single character. /rev3;rev3/

/skip;m1/COMMAND/m2/Marks the beginning and end of a

command string. Initially '^/'.

/skip;m1/DIVIDE/m2/Marks where a word can be divided. If it helps the output line look better, SCRIPT divides the word where this character appears (/rev3/and

NOCASE

No case conversions are done. Lowercase is not converted to uppercase; uppercase is not converted to lowercase.

NUMBER [+ | -] n [,format]

Enables the output of page numbers. If n appears, it sets the page number for the current page (the one on which the number command would have appeared if it were text not in a KEEP or FIGURE.) The new number will be printed in the head of the current page if the number command precedes all text on the page. If a plus or minus sign appears, the current page number is incremented or decremented, respectively. The format, if it appears, is one of 1, I, i, A, or a, to signify arabic numbering, Roman numerals (in upper or lower case), or lettering. The page number format is initially, "1".

/ NØTE/te⁄t / / / / / / / / / / / / / / / / / /
/ /
/ / B⁄ha⁄es/ l⁄ke/ "⁄beg⁄n /no⁄e;/lm⁄15⁄ r⁄m-l⁄; ⁄skip⁄ 2⁄ t⁄ 3⁄
/ / ⁄beg⁄n;/ce⁄te⁄/ ⁄ex⁄ /⁄nd⁄ s⁄kip⁄" /If/th⁄ ⁄ex⁄ ⁄or/ t⁄e /
/ /ti⁄le/do⁄s ⁄not/ap⁄ea⁄, ⁄the/wo⁄d ⁄NO⁄E"/is/ce⁄te⁄ed⁄ ⁄The/
/ / n⁄te/is/en⁄ed/by/"/⁄nd/no⁄e/⁄. /Th⁄re/ i⁄ ⁄no /au⁄om⁄ti⁄
/ / ⁄skip⁄ a⁄ t⁄e ⁄nd/of/th⁄ n⁄te⁄ / / / / / / / / /

PAGE/EPAGE/OPAGE

This command causes a BREAK, forces a switch to column 1 and ends the current page. The foot for the page is printed and the head for the next page is set up to be printed before text is added. If the PAUSE switch is specified, the command rings the console bell and waits for input. Because a NUMBER command may follow the PAGE command, the header is not printed until a complete line is ready for the page. If EPAGE (OPAGE) is executed on an even (odd) page, the following odd (even) numbered page is left blank.

PARAGRAPH [-] [i] [,[s] [,t]]

Behaves like /skip s; testpage t; indent [-] i/. If any of i, s, or t are missing, defaults are used. These

are initially 5, 1, 2, but they can be reset in a block by the AUTOPARAGRAPH command. Notice that the skip s causes a BREAK

* REVISION [date] [,ch] - - - ENDREVISION (ERVSN) [date] [,ch] *
*
* Defines a block of text around which marginal revision *
* "bar"s are to be printed. Within such a block, the *
* character last specified in a REVISION command is *
* printed in the columns established by the VERSION or *
* TEXT command. The initial revision character is "|". *
* NOTE: The revision character is not affected by the *
* nesting of BEGIN-END blocks. A REVISION command toggles *
* the printing of revision characters only if the *
* specified date is the same as or later (greater) than *
* the current VERSION date, or if no date is specified. *
* The date is specified as one to three numbers separated *
* by hyphens, which are interpreted as dd, mm-dd, and mm- *
* dd-yy, respectively. Revision blocks for the same date *
* may not be nested or overlapped. But, revision blocks *
* for different dates may be nested and/or overlapped. Up *
* to 16 revision blocks may be overlapped at any point in *
* the text. (/rev,*/.../erev,!/ brackets this section. *
* Revision characters appear on this page only.) *

RMARGIN [+ | -] n

Sets the right margin to n, unless a plus or minus sign precedes n, in which case the right margin is incremented or decremented by n, respectively. The longest line that will fit between the margins is (rmargin-lmargin) characters long. (Under NOFILL, the right margin is ignored.)

SETTABS [+ | -] n [,] ...

Sets tab stops at the specified positions. A position is relative to the left margin if its specification is a signed number. The tab stops must be given in (absolute) ascending order. Previous tab settings are
! lost (for the current block). Only TAB, MTAB,ITAB,and !
DROP commands in the input cause tabbing in the output line. No tabs stops are set initially.

! The tab stops are numbered 1, 2, 3, ... Tab stop Ø is !
! used to represent "the next tab stop". If the position !
! is unsigned, then the lowest number permitted is 1. !

SPACING n [,T]

Sets the line spacing such that n-1 blank lines are printed between each line generated by the input text.

The initial spacing is 1. If "T" is coded after the spacing value, only lines that have text on them will be followed by the n-1 blank lines. (The "T" mode is useful for generating drafts that will need extensive revisions.)

SPREAD [left] [,[middle] [,right]]

This command causes a BREAK. The left, middle, and right strings are formatted into a line as in the FOOT or HEAD command. The line is then added to the output.

SKIP [n]

This command causes a BREAK, then skips n lines or until the top of a page is reached, whichever comes first. Note that this may be zero lines if already at the top of a page. An exception is made if the top of a page was reached by a PAGE command, so that SKIP immediately following a PAGE will indeed skip. If n is missing, 1 is assumed. Each blank line generated by SKIP is "spaced" under the control of the SPACING command.

/ STOP/ /
/ /
/ / This/command causes/output to stop/at/ the /point/ where
/ / the/ command/ would have/appeared/if/it/were text. /This /
/ /is/useful for changing the/type /element,/ for /example./
/ / It has no effect/unless output is to the/DTC. / / / /

TAB [n]

Tabs to the n-th tab stop. If n is not coded or is Ø, the next tab stop to the right is selected. If the selected tab stop is to the left of the current position, a BREAK is executed, and the new line is started at the tab position. If the current position is the selected tab position, no movement occurs. [Note: /tab/ must be used to tabulate the output text properly. SCRIPT does not recognize the tab character, ^I (Ø9H), as a tab command.]

/ TABLE [/ |/-]/left:/ width, /../ -/- / END / / / / / / /
/ /
/ / Delimits a table./ The TABLE /statement begins a block
/ / and/defines a number of columns by specifying/their/left /
/ /text /margins /and widths/ /If/ a/ plus/ or minus sign/
/ / precedes a/left /margin, /it/ is set/ relative to /the
/ / current left/margin; otherwise/it/is/absolute/ / / / /
/ /
/ / While building /a /table/ SCRIPT/ maintains /a /set of
/ / LMARGIN, RMARGIN values for/ each/ column:/ LMARGIN is /

~~initially zero (which corresponds to the left margin of the column); RMARGIN is initially the column width. The LMARGIN and RMARGIN controls will change these values for the column in which they occur. The table is built from top to bottom and left to right, using the COLUMN command to switch between columns. The columns are numbered from left to right starting at 1. A table may not contain a FIGURE, PAGE, TABLE or KEEP command, but TABLEs may appear within a FIGURE or KEEP.~~

TESTPAGE (TP) t

If fewer than t lines are left on the current page, a PAGE command is executed. This command always causes a break.

TEXT lmargin,width [,bmargin]

This command defines the left margin and width for each text line. After executing /TEXT L,W,B/, LMARGIN zero will correspond to position L+1 of the page, and revision bars will be drawn at physical positions B+1 and W-B. The initial text margins are set by /TEXT 0,85,2/. (At this time, the left margin value is ignored.)

UPPERCASE

In this mode, all lowercase letters are converted to uppercase. This mode is cancelled by the NOCASE command, but may coexist with LOWERCASE for interesting effects. (tHIS SENTENCE WAS TYPED IN NORMAL UPPER/LOWER CASE, BUT GENERATED UNDER /uppercase; lowercase/ CONTROLS.)

USPACE

Flags that spaces are to be underlined whenever underlining is specified. This is the normal condition.

~~NOSPACE~~

~~Flags that spaces are not to be underlined.~~

VERSION [n] [,m]

Sets the document version number (date) to "n" and the bar margin indent to "m". After this command is processed, only REVISION commands referring to VERSION "n" or higher will toggle revision bar printing, and the revision character will be printed in physical positions m and "TEXT width"-m-1 of the output line. The version

number is written in the same form as described in the revision command. A VERSION number of 0 will activate all REVISION blocks in the file, and a VERSION number of 65535 (the largest possible) will deactivate all REVISION blocks.

3. Macros

To enhance the usefulness of SCRIPT, a limited macro processing feature is provided. The user may define parameterized macros that are unconditionally expanded when scanned by SCRIPT. All macro processing is performed before any other SCRIPT command or special character. Thus, a macro expansion may generate SCRIPT commands or special character sequences as well as the text to be included in the output document.

Macro processing is initiated by the scanning of the MACRO flag character. The initial macro flag is "%", but this may be changed by the FLAG MACRO command to any character the user desires. To get the macro flag character through the macro processor without initiating macro processing, the macro flag character must be doubled. (Note: the literalizing flag character will NOT literalize the macro flag character.)

The name of the macro to be processed immediately follows the macro flag. A macro name begins with a letter and continues with any contiguous sequence of digits and upper- or lower-case letters. The case of a letter in a macro name is significant. Some examples are %Chapter, %Date, %Note1, and %Note2.

If the macro name is followed immediately by an "=", a new definition is attributed to the macro name. In its simplest form, the definition is all of the characters between the "=" and the end of the input line. For example:

```
%Date=6/14/82
%Note1=/skip; begin; lm+5; rm-5/ %0 /end; skip/
```

defines two of the above macros, %Date and %Note1.

A macro definition may be continued onto several lines by starting each continuation line with the macro flag followed immediately by an "=". For example, %Chapter and %Note2 are defined as follows:

```
%Chapter=/aux{/t1/%0 /t2/%1 /drop6/ %pn
%=}; skip; tp6/%0.  %1
%Note2=/aux  {%Note1[%1]
%=}/%Note1[%1]
```

When expanded each of these macros will generate two lines of input.

Formal macro parameters are denoted by the flag character followed by a single digit (as %Ø and %1 in the definitions above). Hence a macro may have at most ten formal parameters. Whenever a formal parameter is scanned by the macro processor, the current input source is stacked and the input to SCRIPT is switched to the value of the corresponding actual parameter. When SCRIPT reaches the end of the actual parameter, input is restored to the character following the name of the formal parameter in the macro definition being expanded. Outside a macro expansion a parameter reference is replaced by the null string.

If the macro name is not followed by an "=" the macro name is expanded according to its definition. The expansion proceeds in two steps: first the actual parameters are collected from the input and stored within the macro processor; then the current input source is stacked and input to SCRIPT is switched to the definition of the macro. When SCRIPT reaches the end of the definition, input resumes with the character immediately following the macro call.

The actual parameters to the macro call, if there are any, are enclosed in square brackets immediately following the macro name. The actual parameters are separated by the FLAG DELIMITER character (initially "\"). (Note: an empty parameter list ("[]") may be used to delimit the right end of a parameterless macro call which must be embedded in text.) The only restrictions on the form of an actual parameter are that 1) it must be balanced with respect to square brackets and 2) it must contain no flag delimiter character that is not nested within square brackets. (Note: the macro flag character may be used to literalize an unbalanced square bracket or flag delimiter character.)

As an actual parameter is collected, only formal parameter references within it are expanded; macro calls are not expanded until the actual parameter is substituted into text outside of a macro argument. A formal parameter reference within an actual macro argument is "expanded" by inserting the character sequence currently associated with it directly into the sequence that defines the actual parameter being collected. Nothing is expanded or parsed in this insertion process. Hence, the call to %Note1 within %Note2 is guaranteed to parse correctly regardless of the value of its second parameter (%1).

If the flag macro character is not followed by a letter or digit, the flag macro character serves to literalize it with

respect to macro processing.

Predefined Macros

The only predefined macro is %pn, which always expands to the decimal representation of the current document page number.

B. System Interface

SCRIPT uses the UDI interface to the RUN operating system.

C. Files Used

SCRIPT transcribes text from a list of input files to a single output file. Error messages are directed to the console. The user may optionally generate additional output files using the AUXFILE command. No other files are used.

III. OPERATING SPECIFICATIONS

A. Product Activation Instructions

SCRIPT is invoked by the command:

SCRIPT <input list> [TO <output spec>] {<switch>}

where <input spec> is a list of file specifications (wildcard characters are not permitted), <output spec> is a single file specification (wildcard characters are not permitted), and <switch> is one of the following:

DTC	output is formatted for printing on the DTC-300 terminals. This enables the use of some special characters. Partial spacing is NOT enabled. Pitch is established by the switch on the DTC.
FIRST(n)	The first page numbered n is the first page output.
LAST(n)	The first page numbered n, is the last page output.
PAUSE	SCRIPT types a bell and waits for console input

before printing each output page except the first. This is useful if output on a letterhead, for example, is desired.

PITCH(n) Indicates that partial spacing on the DTC is to be used. Full spacing is set at 60/int(60/n) characters per inch.

/ INDEX / Names a file/or/ files/ to /receive /the/ document /
/ / / /index/ / / / / / / / / / / / / / / /

AUXFILEn = filename
Names the file to receive text sent to auxiliary file n. If the source text uses auxfile i, but no corresponding "AUXFILEi = filename" control was issued in the invocation, then SCRIPT will dynamically create an AUXFILE for i. This file will have the same root as the output file, and the extension will be "AFi".

If SCRIPT is required to generate a default auxfile, then the message

<file> used as auxfilei

will be output to the console.

SCRIPT signs on with the console message:

<system-name> Transcription Service, V1.1

When the output file is opened, SCRIPT issues the console message:

*Transcribing <input file> to <output file>.

When a new input file is opened for the output file, SCRIPT annotates this with the message:

*Transcribing <input file> to <output file>, line #

The message

*Finished in line # of <input file>--<output file> has # lines

announces the end of execution, transcription is complete.

B. Summary of Error Conditions

Any nonzero status returned by a UDI call causes SCRIPT to print the error using the following format:

```
I/O OPERATION -
  FILE:        <file>
  OPERATION:   <script_msg>
  ERROR:       <udi_msg>
```

<File> is the name of the file that the operation was being attempted on. <Script_msg> names the operation SCRIPT was trying to perform. <Udi_msg> is the text returned by the UDI procedure DQ$DECODE$EXCEPTION. Other errors and special situations detected by SCRIPT generate the following messages to the console:

*OUT OF MEMORY in line # of <input file>
SCRIPT does not have enough room for I/O buffers.

*TOO MANY WORDS in output line in line # of <input file>
SCRIPT can't process more than 100 words per line.

*output line TOO LONG in line # of <input file>
The limit is 200 characters per line.

*WORD TOO LONG in line # of <input file>
The limit is 100 characters per word.

*TOO MANY underline flags in line # of <input file>
Only 40 underline flags per line are permitted.

*Undefined Tab or Tab Index in line # of <input file>
In a TAB n command, n is greater than the number of tabs currently set, or in a TAB command, no tabs are defined beyond the current position.

*Too many tab settings in line # of <input file>
The limits is 100 tab stops.

*Tab setting is out of order in line # of <input file>
Tab settings must be specified in ascending order.

*STACK OVERFLOW in line # of <input file>
The stack is used to save command values changed inside a begin block. Each value saved requires 3 or 6 bytes depending on its size. 200 bytes are allocated for the stack.

*NESTING TOO DEEP in line # of <input file>
BEGIN-END blocks may be nested only 10 deep.

*TOO MANY ENDS in line # of <input file>

*TWO PLACES FOR PAGE NUMBER in line # of <input file>
In a head, foot, TOC or SPREAD.

*Unknown COMMAND "<command>" in line # of <input file>

*Unknown FLAG "<command>" in line # of <input file>

*FIGURE nested within KEEP or FIGURE in line # of <input file>

*KEEP nested within KEEP or FIGURE in line # of <input file>

*Unexpected "<command>" in line # of <input file>
Such as an ENDK out of place.

*Unexpected REVISION command in line # of <input file>
Consecutive REVISION commands not separated by an ENDREVISION command.

*TOO MANY (17) active revision levels in line # of <input file>

*Unexpected END REVISION command in line # of <input file>
Consecutive END REVISION commands not separated by a REVISION command.

*MARGINS OVERLAP in line # of <input file>

*Terminating COMMAND FLAG expected in line # of <input file>

*TOO MANY COMMANDS
This is a SCRIPT error and should not appear.

*")" expected following "<switch>"
The parameters on the INDEX, LAST, FIRST, and PITCH switches may be enclosed in parentheses.

*UNKNOWN SWITCH "<switch>"

*incorrect COMMAND LINE format
Invocation line did not end with a CR, LF, ESC or command flag.

*CR not followed by LF in line # of <input file>

*Unbalanced macro parentheses in line # of <input file>

*Undefined macro in line # of <input file>

*Excessive macro arguments in line # of <input file>

*Unexpected HASH command in line # of <input file>

*Unexpected ENDHASH command in line # of <input file>

Appendix I: Single Letter Abbreviations

A	AUTOPARAGRAPH
B	BEGIN (not BREAK)
C	COLUMN
D	DROP
E	END
F	FILL
G	
H	
I	INDENT
J	JUSTIFY
K	KEEP
L	LENGTH
M	MTAB
N	NOAUTOPARAGRAPH
O	OFOOT
P	PARAGRAPH
Q	
R	RMARGIN (not REVISION)
S	SKIP
ST	SETTAB
T	TAB
U	USPACE
V	VERSION
W	
X	
Y	
Z	

Appendix II: Defaults

```
AUTOPARA        (TRUE),  /* INITIAL SPACES CAUSE PARA */
LOWERCASE       (TRUE),  /* CONVERT UCASE TO LCASE */
FORCEUP         (FALSE), /* CONVERT LCASE TO UCASE */
FILL            (TRUE),  /* FILL LINES WITH WORDS */
CENTER          (FALSE), /* CENTER LINES BETW MARGINS */
JUSTIFY         (TRUE),  /* ADD SPACES TO LINE UP RM */
CYCLING         (FALSE), /* CYCLING PAGE NUMBER POS.*/
REVISING        (0),     /* Active REVISION BARS */
REVLINE         (FALSE), /* ADD REVISION BAR TO LINE */
HASHING         (FALSE), /* ADD HASH MARKS TO LINE */
HASHLINE        (FALSE), /* ADD HASH MARKS TO LINE */
TOPPAGE         (TRUE),  /* SITTING AT TOP OF PAGE */
TOPBYCMD        (TRUE),  /* AT TOP CUZ COMMAND */
DTCSW           (FALSE), /* OUTPUTTING TO DTC-300 */
PAUSESW         (FALSE), /* PAUSE BEFORE EACH PAGE */
BROKEN          (TRUE),  /* RECENTLY ISSUED A BREAK */
PAUSE           (FALSE), /* PAUSE AFTER EACH PAGE */
SPACE$NULL      (TRUE),  /* SPACE AFTER NULL LINES */
OUTPUTING       (TRUE),  /* DON'T SUPPRESS OUTPUT */
FIGURING        (FALSE), /* Building a Figure  */
FIGUREBREAK     (0),     /* LINES to BREAK after FIGURE */
KEEPING         (FALSE), /* Building a KEEP block */

LMARGIN         (0),     /* LEFT MARGIN */
INDMARGIN       (0),     /* LEFT MARGIN + INDENT */
RMARGIN         (65),    /* RIGHT MARGIN */
Bar$Margin      (2),     /* Offset for Revision Bar */
Text$Margin     (0),     /* Offset for text */
Hash$Margin     (2),     /* Offset for Hash Marks */
Hash$Offset     (3),     /* Increment between Hash Marks */
Hash$Cycle      (0),     /* Initial Increment for Hash Line */
WIDTH           (85),    /* PAGE/COLUMN  WIDTH */

PAGELEN         (528),   /* NUMBER OF LINES / PAGE */
PAGELINE        (0),     /* NO. LINES ABOVE CURRENT */
HEADMARGN       (16),    /* BLANK BEFORE HEAD */
TOPMARGN        (48),    /* BLANK BEFORE TEXT */
BOTMARGN        (480),   /* BLANK AFTER NORMAL TEXT */
FOOTMARGN       (496),   /* BLANK AFTER FOOT */
PAGENO          (1),     /* PAGE NUMBER */
PAGEFORM        ('1'),   /* FORMAT FOR PAGE NO. */
SPACING         (0),     /* SPACING BETWEEN LINES */
VERSION         (0),     /* DOCUMENT VERSION NUMBER */
HashLevel       (0),     /* Hash Out Level */
Bar$Char        ('|'),   /* Revision Bar Character */
Cmd$Bar$Char    ('|'),   /* Revision Bar Character */
Hash$Char       (' '),   /* Hash Character */
Cmd$Hash$Char   ('/'),   /* Hash Character */
Drop$Char       ('.'),   /* Character for Drop cmd */
```

Appendix II: Defaults (continued)

```
CMDFLAG         ('/'),
LITERALFLAG     ('^'),
ULINEFLAG       ('&'),
MACROFLAG       ('%'),    /* Macro Recognition Char */
DELIMFLAG       ('\'),    /* Macro Argument Delimiter */
NTBFLAG         (7EH),    /* Non-Trivial Blank Char */

AUTOIND         (5),
AUTOSKIP        (1),
AUTOTEST        (2);
```

```
+==================================+
|                                  |
|               MPL                |
|              v1.6                |
|                                  |
+==================================+
```

I. THE MACRO PROCESSOR LANGUAGE

This document describes the syntax and semantics of the Macro Processor Language (MPL).

A. INTRODUCTION

1. GENERAL OPERATION

Understanding MPL requires awareness of the basic algorithm used during its interpretation of textual input. The basic operation performed in all macro processors is substitution in which macro function calls are replaced by their textual values. The processor reads a stream of characters consisting of text that has no meaning to the processor, interspersed with invocations of macro functions. The macro functions guide the process by which textual results are substituted for macro calls. Thus the transformation to the string of characters passing from its input to its output stream.

When the macro processor is in its Normal mode of reading text, the appearance of a distinguished character (the "Metacharacter"-- a "%" will be used in this document) in the input stream will be a signal that a macro processor function is to be invoked. The metacharacter is optionally followed by the call-literally character, "*", which determines how the evaluation is to be performed. Next is the name of a built-in or user-defined macro function to be called. Next, arguments, if any, are identified and evaluated. Finally, the specified function will be performed and the text of the function call will be replaced by the textual value returned by the function. It should be noted that the value may be the null string, and that functions may have side effects. As an example, the function DEFINE has the side effect of causing a macro to be defined. The value of DEFINE is always the null string.

Generally, when the macro processor is in the Literal mode of reading text, the metacharacter is ignored - the text is

treated as literal characters. The exception to this rule is that the three "high priority" functions (Escape, Comment, and Parameter Substitution) are recognized in both Normal and Literal modes.

For a more complete description, see the section entitled "The Macro Processor Scanning Algorithm."

2. TERMINOLOGY AND CONVENTIONS

For the purposes of illustration, a percent sign will be used as the Metacharacter throughout this document. The user may temporarily change the metacharacter by using the METACHAR function.

The term "logical blank" refers to a blank, horizontal tab, carriage return, or linefeed character.

Throughout the document the term "parameter" refers to what are sometimes known as "dummy parameters" or "formal parameters" while the term "argument" is reserved for what are sometimes known as "actual parameters". The terms "Normal" and "Literal", names for the two fundamental modes used by the macro processor in reading characters, will be capitalized in order to distinguish these words from their ordinary usage.

This document uses a BNF style syntactic notation to specify the syntax of the macro processor language. Non-terminal syntactic types are represented by lower case words, sometimes containing the break character, "_". If a single production contains more than one instance of a syntactic type each instance may be followed by more than one unique integer so that the prose description may unambiguously refer to each occurrence.

B. BASIC ELEMENTS OF THE MACRO LANGUAGE

1. IDENTIFIERS

With the exception of some builtin functions, all macro processor functions begin with an identifier, which names the function. Parameters also are represented by identifiers. A macro processor identifier has the following syntax:

```
id = alphabetic | id id_continuation
```

The alphabetic characters include upper and lower case letters. An id_continuation character is an alphabetic character, a decimal digit, or the break character ("_").

Examples: WHILE Add_2 MORE_TO_DO Much_More

An identifier must not be split across the boundary of a macro and may not contain Literal characters.

For example,

"%%(F00)" is illegal. The first metacharacter is followed by the letters "FOO", but they do not constitute an identifier since they are Literal characters.

"%ADD%SUFFIX", where SUFFIX is defined as "UP" is a call to ADD followed by a call to SUFFIX, rather than a call to ADDUP, because identifiers may not cross macro boundaries.

A null-string bracket or escape function ("%()" or "%0") will also end an identifier, and since these functions have no textual value themselves, may be used as separators.

Example:

"%TOM%0SMITH" concatenates the value of the macro, TOM, to the string, "SMITH".

This could also be done by writing, "TOM%(SMITH)".

Upper and lower case letters are equivalent in their use in identifiers. ("CAT", "cat", and "cAt" are equivalent.)

2. TEXT AND DELIMITERS

"Text" is an undistinguished string of characters. It may or may not contain items of significance to the macro processor. In general, the MPL processor simply copies characters from its input to its output stream. This copying process continues until an instance of the metacharacter is encountered, whereupon the macro processor begins analyzing the text that follows.

Each macro function has a calling pattern that must match the text in an actual macro function call. The pattern consists of text strings, which are the arguments to the function, and a number of delimiter strings.

For example, "LEFT(FIRST, SECOND)" might be a

pattern for a macro, LEFT, which takes two arguments. The first argument will correspond to the parameter, FIRST, and the second to the parameter, SECOND. The delimiters of this pattern are "(", ",", and ")".

A text string corresponding to a parameter in the pattern must be balanced with respect to parentheses (see below). A delimiter which follows a parameter in the pattern will be used to mark the end of the argument in an actual call to the macro.

An argument text string is recognized by finding the specific delimiter that the pattern indicates will end the string. A text string for a given argument consists of the characters between the delimiter (or macro identifier) that precedes the text and the delimiter which follows the text.

In the case of built-in functions, there are sometimes additional requirements on the syntax of an argument. For example, the text of an argument might be required to conform to the syntax for a numeric expression.

3. BALANCED TEXT

Arguments must be balanced with respect to left and right parentheses in the usual manner of requiring that the string contain the same number of left and right parentheses and that at no point during a left to right scan may there have been more right parentheses than left parentheses. (An "unbalanced" parenthesis may be quoted with the escape function to make the text meet this requirement.)

4. EXPRESSIONS

Balanced text strings appearing in certain places in builtin macro processor functions are interpreted as numeric expressions:

1. As arguments that control the execution of "IF", "WHILE", "REPEAT", and "SUBSTR" functions.

2. As the argument to the evaluate function, "EVAL".

Expressions are processed as follows:

1. The text of the numeric expression will be expanded in the ordinary manner of evaluating an argument to a macro function.

2. The resulting string is evaluated to both a numeric and character representation of the expression.

Operators (in order of precedence from high to low):

Parenthesized Expressions

```
HIGH    LOW
*   /    MOD   SHL   SHR
+   -
EQ    LT    LE    GT   GE   NE
NOT
AND
OR   XOR
```

Arithmetic:

"17-bit signed" as used by the 8086 Assembler, ASM86

C. INFORMAL DESCRIPTION OF MACRO PROCESSOR FUNCTIONS

Enumerated below are the predefined functions that are built into the Macro Processor Language. Listed is the syntactic type name that will be used later to define the function, an example or schematic of typical usage, and a brief description of what the function does. A more complete description of each function is contained under separate headings of this document.

Three functions, the comment function, the escape function, and the parameter substitution function, may be considered high priority functions because they are generally recognized and evaluated in both the Normal and Literal mode of reading text. The call-literally character may not appear on calls to these functions.

```
comment_function          %'This is a comment'
```

The comment function allows the user to put comments into macro definitions.

```
escape_function           %2((
```

The escape function is used to quote Ø-9 characters (2 in the example) and prevent them from having their ordinary interpretation for the macro processor.

parameter_substitution_function %PARAMETER1

The parameter substitution function is used inside a user-defined macro's body (its definition) to cause the substitution of an argument for a corresponding parameter. It is recognized everywhere except inside a comment or an escape. Macro parameters have exclusive scope, that is, they may be referenced within the body of the macro to which they belong, but not with macros that are called from within that body.

The rest of the functions have no special priority. They are recognized when reading Normally and not recognized when reading Literally:

bracket_function %(Some text)

The bracket function is used to quote some text, i.e., prevent it from being evaluated (except for the comment, escape, and parameter substitution functions). It is always called Literally; the call-literally character is not allowed in the call. This builtin macro function effects a temporary change to the Literal mode from the Normal mode.

evaluate_function %EVAL(12 + %factor * 7)

The evaluate function evaluates text strings which represent numeric expressions. If the argument does not have the syntax of a numeric expression, then an error occurs.

length_function %LEN(ABDEF)

The length function returns a character representation of the number equal to the length of its argument.

equal_function	%EQS(abc, abcd)
greater_than_function	%GTS(abc, abcd)
less_than_function	%LTS(abc, abcd)
not_equal_function	%NES(abc, abcd)
greater_or_equal_function	%GES(abc, abcd)
less_or_equal_function	%LES(abc, abcd)

These functions are used to compare two text strings, using the ordinary "dictionary" ordering of strings. The value is the character representation for minus one if the relation holds, or zero if it does not.

```
macro_def_function      %*DEFINE( MAC P; )  (...%P...)
```

The macro definition function allows the definition of macros which may or may not have parameters. The example shows a schematic definition of a macro, "MAC" with a single parameter, "P", delimited by the character, ";".

```
macro_call_function         %MAC This is an argument;
```

The macro function call causes a previously defined macro to be called.

```
conditional_function  %IF (%VAR EQ Ø) THEN (...)
                                        ELSE (...) FI
repeat_function       %REPEAT (1ØØ)  (...)
while_function        %WHILE (%FLAG)  (...)
exit_function         %EXIT
```

These control functions are analogous to high level language statements of the same names, and have similar functions.

```
in_function           %IN
out_function          %OUT(Message)
ci_function           %CI
co_function           %CO(c)
```

These functions support interactive macro processing by allowing I/O to the console.

```
substr_function    %SUBSTR(abcdef,2,3)
match_function     %MATCH(HEAD,TAIL) (a,b,c,d,e,f,g,h)
```

These functions allow the user to extract subfields from a character string or scan a character string for an occurrence of a particular subfield.

```
metachar_function     %METACHAR(#)
```

This function allows the user to change the character that is to be interpreted as the metacharacter.

D. THE MACRO PROCESSOR SCANNING ALGORITHM

1. LITERAL OR NORMAL MODE OF EXPANSION

At any given time the macro processor is reading text in one of two fundamental modes. When processing of the primary input file begins, the mode is Normal. Normal mode means that macro calls will be expanded, i.e., the metacharacter in the input will cause the following macro function to be executed.

In the simplest possible terms, Literal mode means that characters are read Literally, i.e., the text is not examined for function calls. The text read in this mode is similar to the text inside a quoted character string familiar to most users of high level languages; that is, the text is considered to be merely a sequence of characters having no semantic weight. There are important exceptions to this very simple view of the Literal mode. If the characters are being read from a user defined macro with parameters, the parameter references will be replaced with the corresponding argument values regardless of the mode. The Escape function and the Comment function will also be recognized in either mode.

The mode can change when a macro is called. For user-defined macros, the presence or absence of the call-literally character following the metacharacter sets the mode for the reading of the macro's value. The arguments to a user defined macro are evaluated in the Normal mode but when the processor begins reading the macro's value, the mode changes to that indicated by the call. When the processor finishes reading the macro's definition, the mode reverts to what it was before the macro's processing began.

To illustrate, suppose the parameterless macros, CAT and TOM are defined as follows:

CAT is: "abcd %TOM efgh", and TOM is: "xyz"

Now consider the text fragment,

"... DOG, %CAT, %*CAT, ELEPHANT, ..."

Assume the string is being read in the Normal mode. The first call to CAT is recognized and called Normally. Since the new mode is still Normal, the definition of CAT is examined for macro calls as it is read. Thus the characters "%TOM" in the definition for CAT are recognized as a macro call and again there is a new mode, again also Normal. The

definition for TOM is read, but it contains no macro calls. After the definition for TOM is processed, the mode reverts back to its value in reading CAT (Normal). After the definition of CAT is processed, the mode reverts back to its original value (Normal). At this point, immediately before processing the comma following the first call to CAT, the value of the text fragment processed thus far is:

"... DOG, abcd xyz efgh"

Now the processor continues reading Normally, finally encountering the second call to CAT, this time a Literal call. The mode changes to Literal as the definition of CAT is read.

This time the characters from the definition are read Literally. When the end of the definition of CAT is reached the mode reverts to its original value (Normal) and processing continues. The value of the entire fragment is,

"... DOG, abcd xyz efgh, abcd %TOM, efgh, ELEPHANT, ...".

The use of the call-literally character on calls to builtin macro functions is discussed in the description of each function. The important thing to keep in mind when analyzing how a piece of text is going to be expanded is the environment in which it is read.

2. THE CALL PATTERN

In general, each macro function has a distinctive name which follows the metacharacter (and possibly the call-literally character). This name is usually an identifier, although a few built-in functions have other symbols for names. For identifier named functions, the macro processor allows the identifier to be the result of another macro call.

> For example, suppose the macro, NAME, has the value "BIGMAC" and the macro BIGMAC has the calling pattern, "BIGMAC X & Y;". Then the call,
>
> "... %%NAME catsup & mustard; ..."
>
> is a call to the macro BIGMAC with the first argument having the value " catsup " and the second argument having the value " mustard".

Associated with this name is, possibly, a pattern of delimiters and parameters which must be matched if the macro is to be syntactically correct. The pattern for each

builtin macro function is described in the section of this document dealing with that function. The pattern for a user-defined macro is defined at the time the macro is defined.

At the time of a macro call, the matching of text to the pattern occurs by using the delimiters one at a time, left to right. When a delimiter is located, the next delimiter of the pattern becomes a new goal. The delimiters in the call are separated by either argument text (if there was a corresponding parameter in the macro's definition pattern), or by any number of logical blanks (in the case of adjacent delimiters in the pattern). Arguments are the balanced text strings between delimiters (or possibly between the macro identifier and the first delimiter). Null arguments are permitted.

See the section "Macro Definition and Invocation" for more information on delimiters and their relationship to argument strings.

3. EVALUATION OF ARGUMENTS - PARAMETER SUBSTITUTION

MPL uses "call-by-immediate-value" as the ordinary scheme for argument evaluation. This means that as the text is being scanned for the delimiter which marks the end of an argument, any function calls will be evaluated as they are encountered. In order to be considered as a possible delimiter, characters must all be on the same level of macro nesting as the metacharacter which began the call. In other words, the arguments to a macro can be any mixture of plaintext and macro calls, but the delimiters of a call must be plaintext.

> For example, suppose STRG is defined as "dogs,cats" and MAC1 is a macro with the calling pattern, "MAC1(P1, P2)". Then in the call,
>
> "... %MAC1(%STRG, mouse) ..."
>
> the first argument will be " dogs,cats" and the second argument will be " mouse". The comma in the middle of the first argument is not taken as the delimiter because it is on a different level than the metacharacter which began the call to MAC1.

When all arguments of a macro have been evaluated, the expansion of the body begins, with characters being read either Normally or Literally as discussed under "Literal or Normal Mode of Expansion". One should keep in mind that parameter substitution is a high priority function, i.e.

arguments will be substituted for parameters even if the macro has been called Literally.

II. MACRO PROCESSOR FUNCTIONS

A. THE EVALUATE FUNCTION

The syntax for the Evaluate function is:

```
evaluate_function  =  "EVAL"  "("  expr  ")"
```

The single argument is a text string which will be evaluated as an expression. The character string returned will be the value of the "EVAL" function.

Examples:

```
%EVAL(7)    evaluates to "Ø7H"

%EVAL( (7+3)*2 ) evaluates to "14H"

If NUM has the value "Ø1Ø1B" then
%EVAL( %NUM - 5) evaluates to "ØØH"
```

B. NUMERIC FUNCTIONS: LEN, EQS, GTS, LTS, NES, GES, and LES

These functions take text string arguments and return some numeric information in the form of hexidecimal integers.

```
length_function  =  "LEN"  "("  balanced_text  ")"

equal_function   =
        "EQS"  "("  balanced_text  ","  balanced_text  ")"

greater_than_function =
        "GTS"  "("  balanced_text  ","  balanced_text  ")"

less_than_function =
        "LTS"  "("  balanced_text  ","  balanced_text  ")"

not_equal_function =
        "NES"  "("  balanced_text  ","  balanced_text  ")"
```

```
greater_or_equal_function =
        "GES"  "("  balanced_text  ","  balanced_text  ")"

less_or_equal_function =
        "LES"  "("  balanced_text  ","  balanced_text  ")"
```

The length numeric function returns an integer equal to the number of characters in the text string. The string comparison functions all return the character representation for minus one if the relation between the strings holds, or zero otherwise. These relations are for string compares. These functions should not be confused with the arithmetic compare operators that might appear in expressions. The ASCII code for each character is considered a binary number and represents the relative value of the character. "Dictionary" ordering is used: Strings differing first in their Nth character are ranked according to the Nth character. A string which is a prefix of another string is ranked lower than the longer string.

C. THE BRACKET FUNCTION

The bracket function is used to introduce literal strings into the text and to prevent the interpretation of functions contained therein. (Except the high priority functions: comment, escape, and parameter substitution.) A call-literally character is not allowed; the function is always called Literally.

```
bracket_function  =  "("  balanced_text  ")"
```

The value of the function is the value of the text between the matching parentheses, evaluated Literally. The text must be balanced with respect to left and right parentheses. (An unbalanced left or right parenthesis may be quoted with the escape function.) Text inside the bracket function that would ordinarily be recognized as a function call is not recognized; thus, when an argument in a macro call is put inside a bracket function, the evaluation of the argument is delayed - it will be substituted as it appears in the call (but without the enclosing bracket function).

The null string may be represented as %().

Examples:

```
%(This is a string.)
evaluates to:
"This is a string."

%( %EVAL(10+5) )
evaluates to:
" %EVAL(10+5) "
```

D. THE ESCAPE FUNCTION

The escape function provides an easy way to quote a few characters to prevent them from having their ordinary interpretation. Typical uses are to insert an "unbalanced" parenthesis into a balanced text string, or to quote the metacharacter. The syntax is:

```
escape_function =  /* A single digit, 0 through 9, followed
                      by that many characters.   */
```

The call_literally character may not be present in the call. The escape function is a high priority function, that is one of the functions (the others are the comment function and parameter substitution) which are recognized in both Normal and Literal mode.

Examples:

"... %2%% ..." evaluates to "... %% ..."

"... %(ab%1)cd) ..." evaluates to "... ab)cd ..."

E. MACRO DEFINITION AND INVOCATION

The macro definition function associates an identifier with a functional string. The macro may or may not have an associated pattern consisting of parameters and/or delimiters. Also optionally present are local symbols. The syntax for a macro definition is:

```
macro_def_function =
  "DEFINE" "(" macro_id  define_pattern ")" [ "LOCAL" {id} ]
  "(" balanced_text ")"
```

The define_pattern is a balanced string which is further analyzed by the macro processor as follows:

```
define_pattern =
  {  [{parm_id}]  [{delimiter_specifier}]  }

delimiter_specifier =  /*String not containing non-Literal
                         id_continuation, logical blank, or
                         "@" characters. */

                     |       "@"  delimiter_id
```

The syntax for a macro invocation is as follows:

```
macro_call  =   macro_id  [  call_pattern  ]

call_pattern  =  /* Pattern of text and delimiters
                    corresponding to the definition
                    pattern.  */
```

As seen above, the macro_id optionally may be defined to have a pattern, which consists of parameters and delimiters. The presence of this define pattern specifies how the arguments in the macro call will be recognized. Three kinds of delimiters may be specified in a define pattern. Literal and Identifier delimiters appear explicitly in the define pattern, while Implied Blank delimiters are implicit where a parameter in the define pattern is not followed by an explicit delimiter. Literal delimiters are the most common and typically include commas, parentheses, other punctuation marks, etc. Id delimiters are delimiters that look like and are recognized like identifiers. The presence of an Implied Blank delimiter means that the preceding argument is terminated by the first logical blank encountered. We will examine these various forms of delimiter in greater detail later in this description.

Recognition of a macro name (which uniquely identifies a macro) is followed by the matching of the call pattern to the define pattern. It must be remembered that arguments are balanced strings, thus parentheses can be used to prevent an enclosed substring from being matched with a delimiter. The strings in the call pattern corresponding to the parameters in the define pattern become the values of those parameters.

Reuse of the name for another definition at a later time will replace a previous definition. Built-in macro processor functions (as opposed to user-defined macros) may not be redefined. A macro may not be redefined during the evaluation of its own body. A parameter may not be redefined within the body of its macro.

Parameters appearing in the body of a macro definition (as parameter substitution functions) are preceded by the metacharacter. When the body is being expanded after a call, the parameter substitution function calls will be replaced by the value of the corresponding arguments.

The evaluation of the balanced_text that defines the body of the macro being defined is evaluated in the mode specified by the presence or absence of the call literally character on the call to DEFINE. If the DEFINE function is called Normally, the balanced text is evaluated in the Normal mode before it is stored as the macro's value. If the define function is called Literally, the balanced_text is evaluated Literally before it is stored.

1. LITERAL DELIMITERS

A Literal delimiter which contains id_continuation characters, "@", or logical blanks must be quoted by a bracket function, escape function, or by being produced by a Literal call. Other literal delimiters need not be quoted in the define pattern.

Example 1:

```
  %*DEFINE ( SAY(ANIMAL,COLOR)  ) (THE %ANIMAL IS %COLOR.)
  %SAY(HORSE,TAN)
produces,
  THE HORSE IS TAN.
```

Example 2:

```
  %DEFINE ( REVERSE [ P1  %(.AND.) P2 ] )   (%P2  %P1)
  %REVERSE [FIRST.AND.SECOND]
produces,
  SECOND FIRST
```

2. ID DELIMITERS

Id delimiters are specified in the define pattern by using a delimiter_specifier having the form, "@ id". The following example should make the distinction between literal and identifier delimiters clear. Consider two delimiter_specifiers, "%(AND)" and "@AND " (the first a Literal delimiter and the second an Id delimiter), and the text string,

"... GRAVEL, SAND AND CINDERS ..."

Using the first delimiter specifier, the first "AND",

following the letter "S", would be recognized as the end of the argument. However, using the second delimiter specifier, only the second "AND" would match, because the second delimiter is recognized like an identifier. Another example:

```
Definition:
  %*DEFINE ( ADD P1 @TO P2 @STORE P3. )
  (       LDA  %P1
          MOV  B,A
          LDA  %P2
          ADD  B
          STA  %P3
  )

Macro call:
  %ADD TOTAL1 TO TOTAL2 STORE GRAND.

Generates:
          LDA   TOTAL1
          MOV  B,A
          LDA   TOTAL2
          ADD  B
          STA   GRAND
```

3. IMPLIED BLANK DELIMITERS

If a parameter is not followed by an explicit Literal or Id delimiter then there is an Implied Blank delimiter. A logical blank is implied as the terminator of the argument corresponding to the preceding parameter. In this case any logical blank in the actual argument must be literalized to prevent its being recognized as the end of the argument. In scanning for an argument having this kind of delimiter, leading non-literal logical blanks will be discarded and the first following non-literal logical blank will terminate the argument.

Example:

```
  %*DEFINE ( SAY ANIMAL COLOR  )  (THE %ANIMAL IS %COLOR.)
```

The call,

```
  %SAY HORSE TAN
```

will evaluate to,

```
  THE HORSE IS TAN.
```

In designating delimiters for a macro one should keep in mind the text strings which are likely to appear as arguments. One might base the choice of delimiters for the define pattern on whether the arguments will be numeric,

strings of identifiers, or may contain imbedded blanks or punctuation marks.

The LOCAL option can be used to designate identifiers that will be used within the scope of the macro for local macros. A reference to a LOCAL identifier of a macro occurring after the expansion of the text of the macro has begun and before the expansion of the macro is completed will be a reference to the definition of this local macro. Every time a macro having the LOCAL option is called, a new incarnation of the listed symbols is created. The local symbols thus have dynamic, inclusive scope.

At the time of the call to a macro having locals, the local symbols are initialized to a string whose value is the symbol name concatenated with a unique number. The number is generated by incrementing a counter each time a local declaration is made.

```
Definition:
  %*DEFINE (MAC1  (FIRST,SECOND,THIRD))  LOCAL LABL
  (%LABL:   LHLD   %FIRST
            MOV    A,M
            LHLD   %SECOND
            MOV    B,M
            LHLD   %THIRD
            MOV    C,M
  )

Macro call:
  %MAC1(ITEM,NEXT,ANOTHER)

Generates:(Typically, depending on value for local "LABL")
  LABL03:   LHLD  ITEM
            MOV   A,M
            LHLD  NEXT
            MOV   B,M
            LHLD  ANOTHER
            MOV   C,M
```

F. THE CONTROL FUNCTIONS: IF, REPEAT, and WHILE

These functions can be used to alter the flow of control in a sense analogous to that of their similarly named counterparts in procedural languages; however, they are different in that they may be used as value generating functions as well as control statements.

The three functions all have a "body" which is analogous to the defined value, or body, of a user-defined macro function. The syntax of these functions is:

```
if_function  =  "IF"  "(" expr ")"  "THEN"  "(" body ")"
                          [  "ELSE"  "(" body ")"  ]  "FI"

repeat_function   =   "REPEAT"  "(" expr ")"  "(" body ")"

while_function  =  "WHILE"  "(" expr ")"  "(" body ")"
```

The expression will evaluate to binary numbers. As in PL/M, twos complement representation is used so that negative expressions will map into a large positive number. (e.g., "-1" maps into ØFFFFH.) The bodies of these functions are balanced text strings, and although they look exactly like arguments in the syntax diagrams, they are processed very much like the bodies of user-defined macro functions; the bodies are "called" based upon some aspect of the expression in the IF, REPEAT, or WHILE function. The effects for each control function are described below.

G. THE IF FUNCTION

The first argument is evaluated Normally and interpreted as a numeric expression.

If the value of the expression is odd (=TRUE) then the body of the THEN phrase is evaluated and becomes the value of the function. The body of the ELSE clause is not evaluated.

If the value of the expression is even (=FALSE) and the ELSE clause is present, then the body of the ELSE phrase is evaluated and becomes the value of the function. The body of the THEN clause is not evaluated.

Otherwise, the value is the null string.

In the cases in which the body is evaluated, evaluation is Normal or Literal as determined by the presence or absence of the call-literally character on the IF.

Examples:

```
%IF (%VAL GT Ø) THEN( %DEFINE(SIGN)(1) )
                    ELSE( %DEFINE(SIGN)(Ø) ) FI
```

If the value of the numeric symbol VAL is positive then the SIGN will be defined as "1"; otherwise, it will be defined

as "Ø". In either case, the value of the IF function is the null string.

```
%DEFINE(SIGN) (%IF (%VAL GT Ø)  THEN(1)  ELSE(Ø) FI)
```

This example has exactly the same effect as the previous one.

H. THE REPEAT FUNCTION

The REPEAT function causes its body to be expanded a predetermined number of times. The first argument is evaluated Normally and interpreted as a numeric expression. This expression, specifying the number of repetitions, is evaluated only once, before the expansion of the text to be repeated begins. The body is then evaluated the indicated number of times, Normally or Literally, and the resulting string becomes the value of the function. A repetition number of zero yields the null string as the value of the REPEAT function call.

Examples:

Rotate the accumulator of the 8080 right six times:

```
%REPEAT (6)
 (   RRC
 )
```

Generate a horizontal coordinate line to be used in plotting a curve on a line printer. The line is to be 1Ø1 characters long and is to be marked every 1Ø characters:

```
%REPEAT  (1Ø)  (+%REPEAT (9)  (.))+
```

evaluates to:

```
+.........+.........+.........+....  (etc.) ...+.........+
```

I. THE WHILE FUNCTION

The WHILE function tests a condition to determine whether the body is to be evaluated. The first argument is evaluated Normally and interpreted as a numeric expression.

If the expression is TRUE (=odd) then the body is evaluated, and after each evaluation, the condition is again tested. Reevaluation of the functional string continues until the condition fails (i.e., the value of the expression is an even number.)

The body of the WHILE function is expanded Normally or Literally depending on how the function was called.

Example:

```
%WHILE (%I LT 10)  (  ...
                      ...
                      %IF (%FLAG) THEN(%DEFINE(I) (20)) FI
                      ...
                      ...)
```

J. THE EXIT FUNCTION

The syntax for exit function is:

exit_function = "EXIT"

This function causes termination of processing of the body of the most recently called REPEAT, WHILE, or user-defined macro. The value of the text already evaluated becomes the value of the function. The value of the exit function, itself, is the null string.

Example:

```
%WHILE (%Cond) ( ...
                 ...
                 %IF (%FLAG) THEN (%EXIT) FI
                 ...
                 ...
               )
```

K. CONSOLE INPUT AND OUTPUT

The Macro Processor Language provides functions to allow macro time interaction with the user.

The IN function allows the user to enter a string of

characters from the console. This string becomes the value of the function. The IN function will read one line from the console (including the terminating carriage return line feed).

The OUT function allows a string to be output to the console output device. It has the null string as a value. Before it is written out, the string will be evaluated Normally or Literally as indicated by the mode of the call to OUT.

The syntax of these two functions is:

```
in_function = "IN"

out_function = "OUT" "(" balanced_text ")"
```

Examples:

```
%OUT (Enter the date:)
%DEFINE(DATE)(%IN)
```

The CI function reads a single character from the cold start console. The character becomes the value of the function. CI does not echo the character.

The CO function outputs a single character to the cold start console. It has the null string as a value.

The syntax of these two functions is:

```
ci_function = "CI"

co_function = "CO" "(" char ")"
```

Examples:

```
%CO (A)
%DEFINE(PASSWORD_CHAR)(%CI)
```

L. THE SUBSTRING FUNCTION

The syntax of the substring function is:

```
substr_function =
      "SUBSTR"  "(" balanced_text "," expr1 "," expr2 ")"
```

The text string is evaluated Normally or Literally as

indicated by the mode of the call to SUBSTR. Assume the characters of the text string are consecutively numbered, starting with one. If expression 1 is zero, or greater than the length of the text string, then the value of this function is the null string. Otherwise, the value of this function is the substring of the text string which begins at character number expression 1 of the text string and continues for expression 2 number of characters or to the end of the string (if the remaining length is less than expression 2).

Examples:

```
%SUBSTR (ABCDEFGH,3,4)  has the value "CDEF"

%SUBSTR (%(A,B,C,D,E,F,G),2,100)
                        has the value ",B,C,D,E,F,G"
```

M. THE MATCH FUNCTION

The syntax of the match function is:

```
match function  =
    "MATCH"  "("  id1  delimiter_specifier  id2  ")"
                                    "("  balanced_text  ")"
```

The match function uses a pattern that is similar to the define pattern of the DEFINE function. It contains two identifiers, both of which are given new values as a result of the MATCH function, and a delimiter_specifier. The delimiter_specifier has the same syntax as that of the DEFINE function. The balanced_text is evaluated Normally or Literally, as indicated by the call of MATCH, and then scanned for an occurrence of the delimiter. The algorithm used to find a match is exactly the same as that used to find the delimiter of an argument to a user-defined macro. If a match is found, then id1 will be defined as the value of the characters of the text which precede the matched string and id2 will be defined as the value of the characters of the text which follow the matched string. If a match is not found, then id1 will be defined as the value of the text string, and the id2 will be defined as the null string. The value of the MATCH function is always the null string.

Examples:

Assume XYZ has the value "100,200,300,400,500". Then the

call,

```
    %MATCH(NEXT,XYZ)  (%XYZ)
```

results in NEXT having the value "1Ø0" and XYZ having the value "2Ø0,3Ø0,4Ø0,5Ø0".

```
%DEFINE  (LIST) (FLD1,3E2ØH,FLD3)

%WHILE (%LEN(%LIST) NE Ø)
(    %MATCH(PARM,LIST) (%LIST)
      LDA  %PARM
      MOV  M,A
      INX  H
 )
```

The above will generate the following code:

```
     LDA    FLD1
     MOV    M,A
     INX    H
     LDA    3E2ØH
     MOV    M,A
     INX    H
     LDA    FLD3
     MOV    M,A
     INX    H
```

Assume that SENTENCE has the value "The Cat is Fat." and that VERB has the value "is", then the call,

```
   %MATCH (FIRST %VERB LAST) (%SENTENCE)
```

results in FIRST having the value "The Cat " and LAST having the value " Fat."

N. THE COMMENT FUNCTION

The comment function allows the programmer to comment his macro definition and/or source text without having the comments stored into the macro definitions. The call-literally character may not be present in the call to the comment function. The syntax is:

```
comment_function  =  "'"  text  ( "'"  |  linefeed )
```

When a comment function is recognized, text is

unconditionally skipped until either another apostrophe is recognized, or until a linefeed character is encountered. All text, including the terminating character, is discarded, i.e., the value of the function is always the null string. The comment is always recognized except inside an escape function. Notice that the comment function provides a way in which a programmer can spread out a macro definition on several lines for readability, and yet not include unwanted end-of-line characters in the called value of the macro.

Examples:

```
%' This comment fits within one line.'

%' This comment continues through the end of the line.
```

O. THE METACHAR FUNCTION

The metachar function allows the programmer to change the character that will be recognized by the macro processor as the metachar. The use of this function requires extreme care. The value of the metachar function is the null string. The syntax is:

```
metachar_function = "METACHAR" "(" balanced_text ")"
```

The first character of the balanced text is taken to be the new value of the metachar. The following characters cannot be specified as metacharacters: a logical blank, left or right parentheses, an identifier character, an asterisk, or control characters (i.e., ASCII value < 20H).

III. USING THE MPL MACRO PROCESSOR

A. FURTHER EXAMPLES

The following is intended to provide some further examples and guidance to the user of the MPL processor.

The MPL processor may be viewed as a general "toolbox" for doing textual processing. Before diving into a particular

application one would be well advised to develop a set of tools for the application at hand. A few examples will be examined.

NOTE: the acronym ICAN used below refers to the INTEL 8080 and 8048 assemblers, ASM80 and ASM48.

1. Variations on the DEFINE function

A numeric assignment macro similar to ICAN's SET directive:

```
%*DEFINE (SET A B)  ( %DEFINE (%A)  (%EVAL(%B)) )
```

Example: %SET NUMBER 12+(%VALUE-7)

A macro to concatenate text to a macro time string variable:

```
%*DEFINE (CAT A B)  ( %DEFINE (%A)  (%*%A%(%B)) )
```

Example: %CAT LINE_BUF %(some more chars for buffer)

2. Modifying the Syntax of Other Built-in Macro Functions

It should be obvious from the previous examples that by creating a user-defined macro one can in many cases replace a built-in function with a syntax more suited for the particular application or programming environment.

3. Host Language Extensions

A Conditional Repeat for PL/M:

```
%*DEFINE (REPEATC BODY UNTIL EXPR;) LOCAL XXX
  (%XXX:DO;
     %BODY
     IF NOT( %EXPR ) THEN GOTO %XXX;
   END;
  )
```

Example:

```
%REPEATC
   body_of_the_repeat
UNTIL CHAR0='#' OR DONE;
```

A Macro to Create "messages descriptors" for PL/M (address of character constant, followed by a comma, followed by

length.)

```
%*DEFINE ( MSG(TEXT) ) (.('%TEXT'),%LEN(%TEXT))

Example:        CALL SEND$MESSAGE( %MSG(Hello there.) );
expands to,     CALL SEND$MESSAGE( .('Hello there.'),12 );
```

If it is possible that the argument to MSG could contain the metacharacter, one would substitute "%(%TEXT)" for "%TEXT" in the definition.

B. ICAN TO MPL TRANSLATION

The purpose of this section is to give some indication of how one might go about converting macros for the ICAN processor to those of the MPL processor. This is not intended to be an exhaustive treatment of the subject, but rather a guide to possibilities.

ICAN directives will be considered one at a time, and the analogous MPL syntax will be given along with comments.

1. Arguments

Ordinarily, ICAN uses call-by-name for macro arguments, i.e., the literal text of the macro argument is substituted for the parameter in the body of the macro. Call-by-immediate-value can be forced by preceding the argument by a percent sign, forcing evaluation of the argument before substitution and expansion. MPL uses the exact reverse: call-by-immediate value is the ordinary case, but call-by-name can be achieved by enclosing the argument in the bracket function. In the majority of cases the results are the same; however, it is possible that a user-defined macro in an argument might have different definitions at the point of call and the point of substitution. With this in mind the following transformations are suggested:

ICAN	MPL
argument	%(argument)
%argument	argument

A workable rule might be to make the above transformations only if the argument contains macro calls.

2. Macro Definition and Call

The transformation here is straightforward.

ICAN:

```
macro_id   MACRO   P1,P2,..Pn
           LOCAL   L1
           LOCAL   L2
             ...
           LOCAL   Ln
           body
           ENDM
```

MPL:

```
%*DEFINE (P1,P2,...Pn)
  LOCAL L1 L2 ... Ln
  (body)
```

In transforming user-defined macro calls and parameter references, the metacharacter must be placed in front of the identifiers and the transformation rule for arguments should be followed. Otherwise the call syntax is the same.

3. SET Directive

This is best handled by a user-defined macro that works similarly to ICAN's SET directive.

```
%*DEFINE (SET ID VAL)   (%DEFINE (%ID)  (%EVAL(%VAL)) )
```

The transformation becomes:

ICAN:

```
name     SET     expression
```

MPL:

```
         %SET    name expression
```

4. IF, ELSE, ENDIF Directives

These transformations are very simple:

ICAN:

```
  IF expression        or      IF (expression)
    ...                            ...
  ELSE                         ENDIF
    ...
  ENDIF
```

MPL:

```
  %IF expression       or      %IF (expression)
    THEN ( ... )                 THEN ( ... )
    ELSE ( ... )               FI
  FI
```

5. REPT Directive

Defining a macro similar to ICAN's REPT directive:

```
  %*DEFINE (REPT N BODY @ENDM )
  ( REPEAT (%N)
    ( %BODY
    )
  )
```

The transformation becomes trivial:

ICAN:

```
  REPT n
     ...
  ENDM
```

MPL:

```
  %REPT n
     ...
  ENDM
```

6. IRP Directive

Defining a macro to operate similarly to IRP:

```
  %*DEFINE (IRP  PARM,PLIST BODY @ENDM ) LOCAL LIST
  ( %DEFINE (LIST) (%PLIST)
    %WHILE (%LEN(%LIST) NE Ø)
    ( %MATCH(%PARM,LIST) (%LIST)
      %BODY
    )
  )
```

```
ICAN:

  IRP   dummy,<list>
    ...
  ENDM

MPL: (with above user-defined macro)

  %IRP dummy,%(list)
    ...
  ENDM
```

7. IRPC Directive

Defining a macro to operate similarly to IRPC:

```
  %*DEFINE (IRPC  PARM,TEXT BODY @ENDM ) LOCAL LIST
  ( %DEFINE (LIST) (%(%TEXT))
    %WHILE (%LEN(%*LIST) NE Ø)
    ( %DEFINE (%PARM)
            (%*SUBSTR(%*LIST,1,1)) %'Pick off 1st char.'
      %DEFINE (LIST)
            (%*SUBSTR(%*LIST,2,9999)) %'Save the rest.'
      %BODY
    )
  )

ICAN:

  IRPC   dummy,text
    ...
  ENDM

MPL: (with above user-defined macro)

  %IRPC dummy, text
    ...
  ENDM
```

8. EXITM Directive

The MPL EXIT macro may be substituted directly for the EXITM directive.

```
ICAN                    MPL

  EXITM                   %EXIT
```

IV. STANDALONE MACRO PROCESSOR

A. LINES

A line is a string of characters up to and including a line-feed character. The maximum line length is 255 characters.

B. CONTROLS

Control lines are any lines in the input plaintext whose first character is a "$".

Note that there are no "primary" controls in the standalone version of MPL. Any control can occur anywhere in the file.

$PRINT(filename) (abbreviated PR)
This control causes the current listing file, if any, to be closed and "filename" to be opened as the new listing file. Default is :BB:.

$LIST / $NOLIST (abbreviated LI/NOLI)
Starts / stops the generation of listing to the current listing file. Default is LIST.

$NOGEN (abbreviated NOGE)
The listing file will include lines for the original source only. This is the default listing mode.

$GENONLY (abbreviated GO)
The listing file will include lines for the expanded text only. (Plain text in the original source file plus the values of the macros, but without the text of the macro calls.)

$GEN (abbreviated GE)
The listing file will contain a complete trace of ALL text processed by the macro processor. The value for a macro call will be listed on the line following and will be indented so that the first character of the value appears under the first character of the call. This option produces a very large listing, but if used selectively, it can be useful for debugging macros.

$OBJECT(filename) (abbreviated OJ)
This control causes the current OBJECT file to be closed and "filename" to be opened as the new OBJECT file. This file is the destination of the plaintext and the values generated by macro calls. Default is :CO:.

$OUT / $NOOUT
These controls override the OBJECT control in the same way that LIST and NOLIST override the PRINT control. They can be used to unconditionally turn the flow of characters to the output file on and off.

$INCLUDE(filename) (abbreviated IC)
The console file (:CI:) may be included. Files (except the primary source file) may be included recursively.

$SAVE / $RESTORE (abbreviated SA/RS)
The current settings of the LIST/NOLIST, GEN/NOGEN/GENONLY and OUT/NOOUT switches are saved on a stack. The maximum nesting level of SAVEs is eight. RESTORE resets the switches to the most recent SAVEd values.

$SAVECI(filename)
If :CI: is used as the input file, the text typed in is written to the SAVECI file. This control causes the current save-console-input file, if any, to be closed and "filename" to be opened as the new save-console-input file. Default is :BB:.

$EJECT / $TITLE (abbreviated EJ/TT)
These controls are recognized as legal, but they are ignored.

C. INVOCATION

The invocation line for the standalone version looks like:

```
MPL [ input_file_name [ command_tail ] ]
```

If an input_file_name is not specified in the invocation line, the current console input file will be used as the input file. The output file (OBJECT) is initially :CO:. The PRINT and SAVECI files are initially :BB:.

The command tail is processed just like ordinary macro text before the input file is read. In essence, the command tail is read, and then "INCLUDEd" at the beginning of the primary source file. This is done by writing the command tail to a file called "MPL.TMP" on the same drive as MPL. Note that this does not meet the standard for the way command lines are processed, but has the advantage of allowing any text (including macros) to be used in the command tail. The command tail begins with the first non-blank character after the input file name and may be extended to multiple lines by

making "&" the last character of all except the last line. Controls may be present in the command tail, however note that this necessitates making that first non-blank character a "$", otherwise the command tail will not be treated as a control line.

Example invocations:

```
MPL

MPL  :F2:INFILE.SRC

:F1:MPL :F1:INFIL.SRC $OBJECT(:F1:INFIL.OBJ) PR(:F1:LST) &
   %SET(FLAG,1)  &
   %DEFINE(FOO)(ABC...XYZ) &
   GENONLY
```

D. PREDEFINED SET MACRO

The SET macro is predefined, and may be used as discussed above. Note that the delimiter between the name and expression is a comma, not an implied blank. Also note that since SET is predefined, rather than builtin, the macro name SET may be redefined.

The syntax looks like:

```
    %SET( name, expression )
```

This does exactly the same thing as:

```
    %DEFINE( name ) (%EVAL( expression ))
```

E. NOTE ON CONSOLE INPUT USED AS INPUT FILE

An end-of-file condition for console input can be simulated by the input line "EOF". That is, if the five characters "E","O","F",<cr>,<lf> are read as plaintext from console input, it is considered to be the end of that input file. (Note: the ISIS operating system allows control-z to indicate end-of-file from a keyboard input device.)

F. MACRO ERROR MESSAGES AND RECOVERY

The following is an updated list of macro error numbers with a suggested message. Extra text, passed by the macro processor, is indicated generically by a lower case description following the suggested message. For all of the following errors, the primary error message is followed by a

trace back of the macro stack giving the nesting of pending and active macro calls and INCLUDE's.

100 (64H) - UNDEFINED MACRO NAME
The string following a metacharacter is not a valid macro in this context. The reference is ignored and processing continues with the character following the name.

101 (65H) - ILLEGAL EXIT MACRO
The built-in macro, "EXIT" is not valid in this context. The call is ignored.

102 (66H) - FATAL SYSTEM ERROR
Indicates loss of hardware and/or software integrity discovered by macro processor. MPL closes all files and aborts.

103 (67H) - ILLEGAL EXPRESSION
A numeric expression was required in a builtin macro function, but was ill formed. Abandon the call and continue processing with the character following the illegal expression.

104 (68H) - MISSING "FI"
The IF macro did not have a "FI" terminator. The IF macro is processed normally.

105 (69H) - MISSING "THEN"
The IF macro did not have a "THEN" following the conditional expression clause. The call to IF is abandoned and processing continues at the point in the string at which the error was discovered.

106 (6AH) - ILLEGAL ATTEMPT TO REDEFINE A MACRO: name
Attempted to redefine a built-in, a parameter, or a user defined macro during its expansion.

107 (6BH) - MISSING IDENTIFIER IN DEFINE PATTERN
In a DEFINE, the occurrence of "@" indicated that an identifier type delimiter followed. It did not. The DEFINE is aborted and scanning continues from the point at which the error was detected.

108 (6CH) - MISSING BALANCED STRING
A balanced string, "(...)" in a call to a built-in function is not present. The macro function call is aborted and scanning continues from the point at which the error was detected.

109 (6DH) - MISSING LIST ITEM
In a builtin function, an item of a parenthesized list is missing. The macro function call is aborted and scanning continues from the point at which the error was detected.

110 (6EH) - MISSING DELIMITER
A delimiter required by the scanning of a user defined function is not present. The macro function call is aborted and scanning continues from the point at which the error was detected.

111 (6FH) - PREMATURE EOF
The end of the primary input file was read while the call to a macro was still being scanned.

112 (70H) - DYNAMIC STORAGE (MACROS OR ARGUMENTS) OVERFLOW
Either a macro argument is too long (possibly because of a missing delimiter) or not enough space is available because of the number and size of macro definitions. All pending and active macros and INCLUDE's are popped and scanning continues in the primary source file.

113 (71H) - MACRO STACK OVERFLOW
The macro context stack has overflowed. This stack is 64 deep and contains an entry for each of the following items:

a. Every currently active input file (primary source plus currently nested INCLUDE's).

b. Every pending macro call, that is, all calls to macros whose arguments are still being scanned.

c. Every active macro call, that is, all macros whose values or bodies are currently being read. Included in this category are various temporary strings used during the expansion of some built-in macro functions.

The cause of this error is excessive recursion in macro calls, expansions, or INCLUDE's. All pending and active macros and INCLUDE's are popped and scanning continues in the primary source file.

114 (72H) - INPUT STACK OVERFLOW
The Input stack is used in conjunction with the

Macro stack to save pointers to strings under analysis. The cause and recovery is the same as for Macro stack overflow.

115 (73H) - (Continue Error)
This error never occurs by itself. It is used to pass additional lines of explanatory text (Macro stack trace back) to the driver program after some other error has occurred.

116 (74H) - LONG PATTERN
An element of a pattern, an identifier or delimiter, is longer than 31 characters, or the total pattern is longer than 255 characters. The DEFINE is aborted and scanning continues from the point at which the error was detected.

117 (75H) - ILLEGAL METACHARACTER: character
The METACHAR builtin function has specified a character that cannot legally be used as a metacharacter: a blank, letter, numeral, left or right paren, or asterisk. The current metacharacter remains unchanged.

118 (76H) - UNBALANCED ")" IN ARGUMENT TO USER DEFINED MACRO
During the scan of a user defined macro the parenthesis count went negative indicating a unmatched right parenthesis. The macro function call is aborted and scanning continues from the point at which the error was detected.

119 (77H) - ILLEGAL ASCENDING CALL
A macro call beginning inside the body of a user-defined or builtin macro was incompletely contained inside that body, possibly because of a missing delimiter for the macro call.

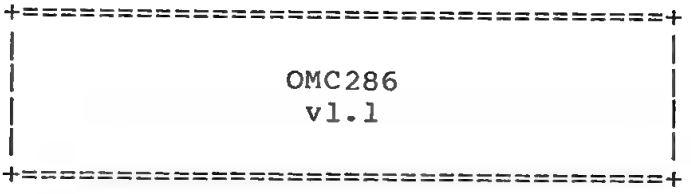
In addition to the above errors the following errors are detected in the dollar-sign control lines which are scanned by the macro processor:

200 - BAD CONTROL PARAMETER

201 - MULTIPLE INCLUDES ON ONE CONTROL LINE

202 - NON IDENTIFIER FOUND IN PLACE OF CONTROL

INVALID COMMAND: bad_command



OMC286 will convert an object module conforming to the 8086 object module formats to a valid linkable object module in the 80286 object module formats. Existing 8086 programs or libraries, in which the source is not available or the user does not wish to look at the source, can be converted to linkable 80286 object modules, without recompilation of the source. Converted object modules can then be used as input to 80286 utilities, such as BND286, BLD286 and LIB286.

Acceptable Input

1) Output of 8086 translators, which do not contain absolute addresses. (e.g. AT(@Abs_address) in PL/M86)

2) Partially linked modules (output of Link86), which do not contain overlays or absolute addresses.

3) Modules created by LINK86 using the BIND control.

Unacceptable Input

1) Libraries, except as specified below

2) Modules containing overlays

3) Modules which contain absolute addresses

4) Output of LOC86

5) Object modules containing interrupt procedures

Any unacceptable input will cause a fatal error and OMC286 processing will be aborted.

Invocation Command

[[device]RUN] [device]OMC286 inputfilename [controls]

Inputfilename is the pathname of an 8086 object file.

controls: ERRORPRINT [(pathname)] | NOERRORPRINT

OBJECT [(pathname)] | NOOBJECT

Each of the OMC286 controls has an abbreviated form. If a keyword has the prefix 'NO' then the corresponding abbreviated form also has the prefix 'NO'.

ERRORPRINT	EP
OBJECT	OJ

Default controls: NOERRORPRINT
OBJECT (inputfilename.OMC)

Controls

If there are multiple specifications of the same control in the invocation command line then the last instance of that control will remain effective during the execution of the OMC286 utility.

ERRORPRINT and NOERRORPRINT

The ERRORPRINT control with a path_name (e.g., "ERRORPRINT(:F1:PROG.ERR)") directs all the error messages (warnings, errors, and fatal errors) to the indicated file. The ERRORPRINT control without a path_name (e.g., "ERRORPRINT") directs all the error messages to the default error print file ":CO:".

NOERRORPRINT control suppresses direction of error messages to the error print file. Fatal error messages will appear on the output console regardless of NOERRORPRINT. NOERRORPRINT is the default.

OBJECT and NOOBJECT

The OBJECT control designates the path_name to which the output module is to be written. This file may not appear as the input file, and must be a file randomly accessible (e.g., a disk file). If the OBJECT control is not specified in the command tail, then OMC286 will generate an output object file which has the same name as the input object file, with the extension ".OMC". The specification of the OBJECT control without a path_name will also have the same result.

Example Invocation

```
RUN OMC286 :F1:PROG.OBJ ERRORPRINT(:F1:PROG.ERR)
```

This invocation creates the file :F1:PROG.OMC as the output file. All warnings and error messages are directed to the file :F1:PROG.ERR.

Sign-on and Sign-off

OMC286 will always sign on to the system console with the following message:

```
system-id iAPX286 Object Module Converter, Vx.y
Copyright 1983 Intel Corporation
```

Completion of OMC286 processing will be indicated by a sign-off message sent to the system console. If there are no fatal errors OMC286 will sign-off as shown below:

```
PROCESSING COMPLETE.  n WARNINGS, m ERRORS
```

The n and m represent the number of warning and nonfatal error conditions, respectively, encountered during processing. The ERRORPRINT control may be used to direct OMC286 to display warning and error messages at the console.

If OMC286 encounters a fatal error condition, the sign-off message will be:

```
PROCESSING ABORTED
```

Fatal error messages are always displayed at the console.

Usage of OMC286

1) Output modules created by 8086 translators can be individually converted into 80286 OMFs and then used as input to any of the 80286 utilities.
2) Multiple modules created by 8086 translators can first be prelinked using LINK86 with its NOBIND control. Then that output module may be converted to 80286 OMFs using OMC286.
3) Output of Link86 under BIND control may be converted into an 80286 OMF format. In this case, the converted output must be run through BND286, with no additional object modules and the LOAD control must also be used.

Examples of Usage of OMC286

Assume there exists a program with 8086 object modules called MAIN.OBJ and UTIL.OBJ. There is also a file called MAIN.86, which is the result of:

```
LINK86 MAIN.OBJ, UTIL.OBJ TO MAIN.86 BIND
```

Here are three ways the user can convert this program to a loadable 80286 program, corresponding to the three ways to use OMC286 above.

```
1) RUN OMC286 MAIN.OBJ
   RUN OMC286 UTIL.OBJ
   RUN BND286 MAIN.OMC, UTIL.OMC

2) RUN LINK86 MAIN.OBJ, UTIL.OBJ TO MAIN.LNK
   RUN OMC286 MAIN.LNK
   RUN BND286 MAIN.OMC

3) RUN OMC286 MAIN.86
   RUN BND286 MAIN.OMC
```

Converting 8086 Libraries

Assume there exists an 8086 library called UTIL.LIB, that contains the two object modules UTIL1_MOD and UTIL2_MOD, to be converted to an 80286 library. If the names of the modules in the library to be converted are not known, use LIB86 with the LIST command to find out the names of the modules in the library. Next use LINK86 to separate each module from the library.

```
RUN LINK86 UTIL.LIB( UTIL1_MOD ) TO UTIL1.86
RUN LINK86 UTIL.LIB( UTIL2_MOD ) TO UTIL2.86
```

Now use OMC286 to convert the 8086 object files.

```
RUN OMC286 UTIL1.86 OBJECT(UTIL1.286)
RUN OMC286 UTIL2.86 OBJECT(UTIL2.286)
```

Then use LIB286 to create an 80286 library containing the modules UTIL1.286 and UTIL2.286.

DEBUG Information

OMC286 will not convert any debug information existing in the input module to the output module. However, public and external symbols in the input module will be converted and transferred to the output module. These symbols will then be configured into debug information (80286 style) by the 80286 utilities (since the output module created by the OMC286 will be an 80286 style non-prelinked module).

Problems Due to Invalid Code Sequences

These problems occur due to some unique coding practices which are valid while executing on 8086 but not on 80286. An examples of such code is loading a segment register (particularly the ES register, which is considered to be a "volatile" register) with some invalid value. On 8086, such practices are legal and are indeed used. However, on 80286, the processor will generate a hardware fault under such conditions. The OMC286 utility will not look at the code sequences in the input modules and will therefore be unable to detect such situations. Moreover, such situations may occur during execution over which no utility can have any control. The only way out of these situations is for the users to change the source appropriately so that the programs still work, but do not use any invalid code sequences.

Similar cases may exist in programs using 8087 instructions and will have to be resolved by the users. Moreover, if programs containing floating point instructions are prelinked with the 8087-emulator library and then converted to the 80286 OMFs, then the output module may not be executable using the 80287 processor. It is therefore advisable that, modules containing floating point instructions be converted to 80286 OMFs without prelinking with the 8087 emulator library. Note that, the OMC286 utility will not attempt to detect whether an input module is a prelinked module or not.

Fragmentation of Logical Segments

Fragmentation of logical segments can occur only when the segments are part of a group.

Note that if each individual module of 8086 format is converted into an 80286 module separately, then in each conversion, the specific meaning associated with the input

segments such as "CONST", "CODE", "DATA" etc. will be lost in the output module. Therefore, when multiple converted modules are further "bound" by the 80286 utilities, each individual "CONST", "CODE" and "DATA" segment will be physically separated from the other, although they will still be part of the same segment, CODE or DATA (or whatever). This is only a cosmetic problem since references to symbols within each of these segments will be guaranteed to be equivalent to those in the input. The cosmetic problem may arise when the users try to disassemble some code which may be followed by a CONST portion.

Support for Floating Point Instructions

An 8086 program, using floating point, always contains a "wait" instruction before each floating point instruction; this wait instruction is not always needed in 80286. OMC286 will leave them as they exist in the input module. This is not expected to cause any problems.

Executable Segment Names

The following is a list of segment names that become executable 80286 segments, when converted by OMC286.

```
CODE
*_CODE          /* "*" is any character string */
DCONCODE
LIB_E87_PUB
LIB_E87_INT
LIB_E87_INIT
LIB_E87_INTP
LIB_E87_INITP
UTSCODE
MQ_CEL_CODE
DECODE_CODE
ENCODE_CODE
SIEVE_CODE
NORMAL_CODE
LIB_87N_PUB
LIB_87_NULL
LIB_87_NULLP
LIB_87_PUB
LIB_87_INIT
LIB_87_INITP
```

Cautions

Converting Main Modules

8086 compilers always create an STI instruction for main modules. STI is a privileged instruction on the 80286. Use the TASK PRIVILEGE control with BND286, in order to get the converted main module to run on the 80286.

Programs already Linked with System Dependent Libraries

Programs already linked with UDI, I/O libraries or system dependent libraries, may not run on an 80286-based system since these libraries may be different in the two environments. Therefore, these programs should be converted without these libraries.

Programs Using Numeric Libraries

8086 programs which use the 8087 emulator (E8087.LIB and E8087) cannot be converted as they are. Consequently, they must be modified to use the 8087.LIB instead of the E8087.LIB and E8087. 8086 programs which use 8087.LIB must link in 8087.LIB before conversion. All public symbols should be purged (using LINK86's NOPUBLICS control) once the 8087.LIB has been linked with the program to be converted. All other numeric libraries required by the program, must be 80287 versions of the libraries in order for the converted program to run.

PASCAL Programs

8086 Pascal programs, using WRITE, WRITELN, READ or READLN will execute on 80286 if linked with 80286 pascal run-time libraries. REWRITE and RESET will work if the second parameter, to these procedures is also specified in the source. NEW and DISPOSE will work if the converted program is linked with 80286 Pascal run-time libraries. Programs using the default "Input/Output" scheme will not execute on the 80286 although they will be converted by OMC286 without any warnings and/or errors. When using BND286 to link the converted program, increase the LDTSIZE by at least 15.

General:

Programs containing interrupt procedures cannot be converted.

Public symbols on STACK segments are not supported on the 80286.

The method used to obtain stackpointer in ASM86 programs (i.e., defining a label beyond the segment limit) is invalid on 80286. BND286 will issue an error if it detects such programming practice.

Warning and Error Messages

SYSTEM INTERFACE ERROR
error text
FILE: pathname

MEANING: A fatal error occurred in a call to the host operating system. The error text contains a message issued by the operating system. The pathname is present if the error is an I/O error.

CAUSE: Such problems as an I/O error, or invalid parameters can cause this condition.

EFFECT: OMC286 processing is aborted and control returned to the operating system.

ACTION: Correct the error and restart OMC286.

ERROR 100: INPUT FILE MISSING

MEANING: No input file name appeared on the invocation line.

EFFECT: OMC286 processing is aborted and control returned to the operating system.

ERROR 101: PATHNAME TOO LONG
NEAR: token string

MEANING: This fatal error occurred because there are too many characters in a pathname in the invocation line near the token string.

EFFECT: OMC286 processing is aborted and control returned to the operating system.

ACTION: Re-invoke OMC286 using a valid pathname.

ERROR 102: MISSING LEFT PARENTHESIS
NEAR: token string

MEANING: This fatal error occurred because a left parenthesis is missing after the token string.

EFFECT: OMC286 processing is aborted and control returned to the operating system.

ACTION: Insert a left parenthesis in the proper location and re-invoke OMC286.

ERROR 103: MISSING RIGHT PARENTHESIS
NEAR: token string

MEANING: This fatal error occurred because a right parenthesis is missing after the token string.

EFFECT: OMC286 processing is aborted and control returned to the operating system.

ACTION: Insert a right parenthesis in the proper location and re-invoke OMC286.

ERROR 104: FILE ALREADY SPECIFIED IN COMMAND TAIL
FILE: pathname

MEANING: This fatal error occurred because the pathname exists in more than one place in the invocation line. One of the duplicate pathnames is explicit in or implied by the controls.

EFFECT: OMC286 processing is aborted and control returned to the operating system.

ACTION: Re-invoke OMC286, ensuring that none of the pathnames, explicit or default, match.

ERROR 105: INVALID DELIMITER IN COMMAND TAIL
NEAR: token string

MEANING: This fatal error occurred because the invocation line contained an improperly placed delimiter or used an illegal character as a delimiter. The invalid delimiter was detected either before or after the token string.

EFFECT: OMC286 processing is aborted and control returned to the operating system.

ACTION: Re-invoke OMC286 using valid delimiters in the invocation line. Valid delimiters include left and right

parentheses and commas.

ERROR 107: UNKNOWN CONTROL IN COMMAND TAIL
NEAR: token string

MEANING: This fatal error occurred because one of the controls in the invocation line is invalid. The invalid control is contained in token string.

EFFECT: OMC286 processing is aborted and control returned to the operating system.

ERROR 108: SYNTAX ERROR

MEANING: This fatal error occurred because the structure of the invocation line near token string is incorrect.

EFFECT: OMC286 processing is aborted and control returned to the operating system.

ERROR 109: CANNOT OPEN WORKFILE FOR SYMBOLS

MEANING: This fatal error indicates that OMC286 cannot create a temporary file to use.

EFFECT: OMC286 processing is aborted and control returned to the operating system.

ACTION: Ensure all temporary files created by the operating system are deleted, then re-invoke OMC286.

ERROR 110: NUMBER OF SYMBOLS EXCEEDS INTERNAL LIMIT

MEANING: This fatal error occurred because the maximum number of symbols OMC286 can process has been exceeded.

EFFECT: OMC286 processing is aborted and control returned to the operating system.

ERROR 111: NOT ENOUGH DISK SPACE FOR SYMBOLS

MEANING: This fatal error occurred because OMC286 does not have sufficient disk space to create internal data structures.

EFFECT: OMC286 processing is aborted and control returned to the operating system.

ACTION: Allocate files across disks in such a way that OMC286 has more space for temporary storage (refer to operating system documentation).

ERROR 112: INVALID OBJECT FILE
FILE: pathname
MODULE: module name

MEANING: This fatal error indicates that the module name, contained in the file referred to by pathname, has an invalid format.

EFFECT: OMC286 processing is aborted and control returned to the operating system.

ERROR 121: NOT AN OBJECT FILE
FILE: pathname

MEANING: The input filename on the invocation line does not contain an 8086 object file. The file could be a library file.

EFFECT: OMC286 processing is aborted and control returned to the operating system.

ACTION: If the input file was an 8086 library file, the library can be converted by using OMC286 to convert each object file which makes up the library separately. Then use LIB286 to regenerate the library.

ERROR 122: OBJECT FILE IS ABSOLUTE
FILE: pathname
MODULE: module name

MEANING: Input file contains an 8086 object file which is absolute. This file is not acceptable input to OMC286.

EFFECT: OMC286 processing is aborted and control returned to the operating system.

ERROR 123: OBJECT FILE CONTAINS OVERLAYS
FILE: pathname
MODULE: module name

MEANING: Input file contains overlay information. Probably the input file is the output of LINK86 using the OVERLAY control. This file is not acceptable input to OMC286.

EFFECT: OMC286 processing is aborted and control returned to the operating system.

ERROR 124: OBJECT FILE CONTAINS AN ILLEGAL RECORD
FILE: pathname
MODULE: module name

MEANING: Input object file contains a record that indicates that this file is not acceptable input to OMC286. The file could be an absolute file or a file that contains an interrupt procedure. Neither is acceptable as an input file to OMC286.

EFFECT: OMC286 processing is aborted and control returned to the operating system.

WARNING 125: REGISTER INITIALIZATION FOR ES
FILE: pathname
MODULE: module name

MEANING: Input object file contains register initialization for ES register. This information cannot be converted.

EFFECT: OMC286 continues processing. Output module does not contain ES register initialization.

WARNING 126: SYMBOL TABLE SPILLS TO SECONDARY STORAGE

MEANING: This warning indicates that OMC286 is using disk space for symbol information, since available memory has already been used.

EFFECT: OMC286 processing speed is reduced.

ERROR 127: ILLEGAL ABSOLUTE SEGMENT
FILE: pathname
MODULE: module name

MEANING: Input object file contains a segment which is absolute. File could be output of LOC86.

EFFECT: OMC286 processing is aborted and control returned to the operating system.

ERROR 128: ILLEGAL ABSOLUTE RECORD
FILE: pathname
MODULE: module name

MEANING: Input object file contains an absolute record. OMC286 will not process absolute files.

EFFECT: OMC286 processing is aborted and control returned to the operating system.

ERROR 129: TYPE DESCRIPTION TOO LONG
FILE: pathname
MODULE: module name

MEANING: OMC286 encountered a type record that was too long to be processed by OMC286.

EFFECT: OMC286 processing is aborted and control returned to the operating system.

ACTION: To convert this file, relink using LINK86 with the NOTYPE control. Then run the output of LINK86 through OMC286.

WARNING 130: SIZE OF GROUP EXCEEDS 64K
GROUP: groupname

MEANING: A group has been found whose component segments do not all lie within the 64K physical segment defined by the group base.

EFFECT: OMC286 processing continues. When using BND286 on the output of OMC286 addressing errors may result.

ERROR 131: INVALID OVERLAPPING GROUPS
SEGMENT: segment name
GROUP: groupname

MEANING: The same segment appears in more than one group.

EFFECT: OMC286 processing is aborted and control returned to the operating system.

ERROR 132: ILLEGAL FIXUP
FILE: pathname
MODULE: module name

MEANING: Input file contains a fixup which cannot be processed by OMC286.

EFFECT: OMC286 processing is aborted and control returned to the operating system.

WARNING 133: EXTRA REGISTER INITIALIZATION INFORMATION IGNORED

MEANING: Register initialization appeared more than once in the input file.

EFFECT: OMC286 continues processing. OMC286 uses the first occurrence of the register initialization in the output file.

WARNING 134: CS REGISTER NOT INITIALIZED

MEANING: No initialization for the CS register is found in the input file which is a main module.

EFFECT: OMC286 continues processing. CS is initialized to zero.

WARNING 135: SS REGISTER NOT INITIALIZED

MEANING: No initialization for the SS register is found in the input file which is a main module.

EFFECT: OMC286 continues processing. SS is initialized to zero.

WARNING 136: DS REGISTER NOT INITIALIZED

MEANING: No initialization for the DS register is found in the input file which is a main module.

EFFECT: OMC286 continues processing. DS is initialized to zero.

WARNING 137: CANNOT GENERATE STACK SEGMENT

MEANING: An LTL input file contains a group called DGROUP, which contains a segment called STACK. OMC286 is unable to convert DGROUP into the two segments DATA and DATA with the STACK attribute, because the size of DGROUP is 64K.

EFFECT: OMC286 continues processing. No STACK segment exists in the output file.

ACTION: Relink the input file, using LINK86 with the SEGSIZE control, to reduce the size of the STACK segment.

WARNING 138: SIZE OF MEMORY SEGMENT REDUCED

MEANING: An LTL input file contains a group called DGROUP, which contains a segment called STACK. The size of DGROUP is 64K. OMC286 reduces the size of the memory segment in DGROUP so that the 80286 segment called DATA with the STACK attribute can be created. This condition can occur for input files that were the result of LINK86 BIND of input files using the SMALL or MEDIUM model of computation.

EFFECT: OMC286 continues processing. Size of memory is reduced.

WARNING 139: SIZE OF STACK REDUCED

MEANING: Input file contains a group called DGROUP, which contains a segment called STACK. If OMC286 created a segment DATA with the STACK attribute the size of the input STACK in DGROUP, the 80286 segments DATA plus DATA with the STACK attribute would be greater than 64K. OMC286 creates a smaller stack segment. This condition can occur for input files that were the result of LINK86 BIND of input files using the SMALL or MEDIUM model of computation.

EFFECT: OMC286 continues processing. Size of stack is reduced.

WARNING 140: SIZE OF MEMORY SEGMENT REDUCED ZERO

MEANING: An LTL input file contains a group called DGROUP, which contains a segment called STACK. The size of DGROUP is 64K. The size of the STACK in DGROUP is greater than the size of MEMORY in DGROUP. OMC286 reduces the size of the memory segment in DGROUP to zero so that DATA STACK can be created. This condition can occur for input files that were the result of LINK86 BIND of input files using the SMALL or MEDIUM model of computation.

EFFECT: OMC286 continues processing. Size of memory is reduced to zero.

ACTION: If memory is needed to execute the program, relink the input file with LINK86, using the SEGSIZE control to decrease the size of the STACK segment.

Conversion of Segmentation Models

This section requires the knowledge and understanding of segmentation models.

The users should be cautioned that programs which "mix" different segmentation models (e.g., SMALL and COMPACT) should be converted with proper understanding of how the segments will be placed after the conversion. In this case, the users may have to prelink some modules using LINK86 and then convert that output to 80286 OMFs. This is particularly true for the MEMORY segment in the SMALL/COMPACT/MEDIUM models.

First of all, it must be noted that the segmentation model

as represented by the input object module must be deciphered so that a meaningful conversion to 8Ø286 OMFs can be made. This is mainly due to the convention of the (DATA+STACK) combined segments and the absence of GROUPs in 8Ø286 OMFs. This is necessary only in the case of SMALL and MEDIUM models.

Conversion of DGROUP

DGROUP gets converted to an 8Ø286 segment called DATA. If DGROUP contains a segment called STACK, which is the case using the SMALL or MEDIUM segmentation models, another 8Ø286 segment is created. This segment is also called DATA but has the STACK attribute.

Since the MEMORY segment will become part of the 8Ø286 DATA segment, users of OMC286, which depend upon the memory segment to be at the top of memory as well as to be of non-zero length may have some problems.

The following solutions can be used to solve this problem:

- The users can ensure that the MEMORY segment will have enough size before it is converted by OMC286, thereby alleviating the problem caused by its indeterminate length at the time of execution. Users can ensure such size by prelinking the modules using LINK86 utility with its BIND and SEGSIZE control, or by writing an ASM86 module which does nothing but to specify the size of the MEMORY segment. If the user wishes to combine the output of OMC286 with other modules, an ASM86 module, which specifies the size of the memory segment will have to be used. The memory segment size (set by the assembly language program) should be made as large as possible, especially if the program being converted uses DqGetSize and .MEMORY to find the memory segment size.

  Example ASM86 program to set the memory segment size.

```
name mem
memory segment word memory 'memory'
memsize  db ØF6ØØH dup (?)
memory ends
end
```

  ØF6ØØH is the memory size in the example above.

- If the user also requires that the MEMORY segment be at the top of the 8Ø286 DATA segment (besides its being of some specific size) then the module containing the

nonzero length MEMORY segment should be specified as the last element in the input list to the 80286 utilities (BND286 or BLD286). This is under the assumption that the DGROUP of the corresponding 8086 module has the MEMORY segment at the top in the first place (this is the High Level Language's default).

Conversion of DGROUP for LTL Modules

In this case, the offsets of symbols and references to them have already been calculated by LINK86 and can not be changed. DGROUP gets converted to an 80286 segment called DATA. If DGROUP contains a segment called STACK, which is the case in SMALL and MEDIUM segmentation models, another 80286 segment is created. This segment is also called DATA but has the STACK attribute. The segment called DATA with the STACK attribute will be created with the same size as the 8086 STACK segment in DGROUP if possible. The amount of space the 8086 STACK took in DGROUP becomes wasted space when DGROUP is converted. If the size of DGROUP plus the size of the STACK segment in DGROUP is greater than 64K, OMC286 makes some adjustments so that the two 80286 DATA segments created will not be greater than 64K.

If the size of DGROUP plus the size of the STACK segment in DGROUP is greater than 64K and the size of DGROUP is less than 64K, then the size of the 80286 STACK (segment called DATA with the STACK attribute) will be less that the 8086 STACK. OMC286 will generate a warning.

If the size of DGROUP is exactly 64K, OMC286 creates the 80286 STACK (segment called DATA with the STACK attribute) by reducing the size of the 8086 MEMORY segment. OMC286 will generate a warning that the size of memory is reduced.

If the resulting size of the 80286 stack and memory area is not sufficient to make the program executable, run LINK86 again on the input module to OMC286, using the SEGSIZE control to adjust the sizes of the segments STACK and MEMORY.

Conversion of CGROUP

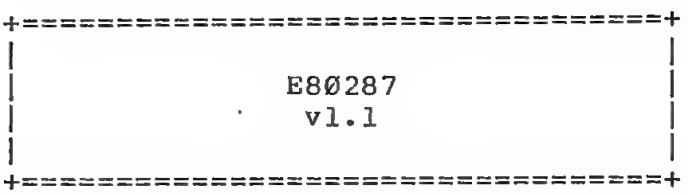
An 8086 group named CGROUP will be converted to an 80286 segment called CODE. An 8086 group named *_CGROUP (* means any name) will be converted to an 80286 segment called *_CODE.

Conversion of Other 8086 Groups

All other 8086 groups will be converted to an 80286 segment with the same name as the 8086 group. If the group contains an executable segment then the group will be converted to an 80286 segment with access type execute read; otherwise the group will be converted to an 80286 segment with access type read write.

Conversion of 8086 Segments not Part of Groups

8086 segments not part of groups will be converted to one 80286 segment with the same segment name. Segments with the name CODE or *_CODE will become executable 80286 segments. All of the converted segments will have a privilege level of 3.



E80287 is a software emulator for the 80287 numeric data processor. It is intended to serve environments which require floating point processing, but which lack the 80287 NDP. Any iAPX-286 program that contains floating point operations may be executed on either an 80287 component or E80287. The following paragraphs describe the usage and characteristics of the emulator.

Requirements

The emulator must reside in memory with the floating point programs which use it. It occupies about 10 kbytes of memory, including one data segment of 94 bytes and another of 2 bytes. A STACKSEG directive reserves 160 bytes of the system stack. The emulator has a conforming readable code segment which may be stored in ROM. The processor must operate in the protected virtual address mode while executing E80287. Programs which use E80287 must be located in readable code segments. Floating point instructions invoke E80287 by means of the iAPX-286's processor extension trap mechanism. Before any task uses the emulator, the processor extension trap must be initialized.

Initialization

The system which uses the 80287 emulator must execute an initialization procedure whenever a copy of E80287 is loaded to memory. The emulate processor extension control of the iAPX-286's machine status word must be set, and the emulator's entry point, "E80287", must be stored into the interrupt 7 descriptor of the IDT. An E80287 software library has been included in the toolbox to assist with initialization. Both E80287 and its library, E80287.LIB, should be bound to the task which performs initialization. A single call to the FAR procedure XQ_INITE287 will set the machine status control and load the emulator trap pointer. Because it contains privileged instructions, XQ_INITE287 must execute at privilege level 0. A second library procedure called XQ_SET_IDT may be used to set other entries in the interrupt descriptor table, such as the floating point exception descriptor. Calls to this FAR procedure take the form of the following PL/M statement:

```
CALL XQ_SET_IDT(IDT_ENTRY,GATE_DPL,PROC_PTR);
```

where IDT_ENTRY is the interrupt number, GATE_DPL is the interrupt gate data privilege level, and PROC_PTR points to the interrupt procedure. As long as the same copy of E80287 remains at the same location, it need not be re-initialized and may be executed by tasks at any privilege level.

Re-entrancy

The 80287 emulator is designed to be completely re-entrant and easily multitasked. All temporary variables are stored in the current stack frame which is allocated and deallocated after every instruction trap. The context of the emulated 80287 is stored in the data segment A?MSR between floating point operations. As long as the contents of this data segment are preserved, any number of tasks may use E80287 concurrently. The 80287 context may be switched in the conventional manner, using FSAVE and FRSTOR instructions, but there is an even faster method. The data segment A?MSRS contains the single PUBLIC WORD PTR A?MSR_SELECTOR. This variable defines the selector of the data segment A?MSR for E80287. All internal emulator references to A?MSR are made through A?MSR_SELECTOR. If the selector is reassigned for each new task that uses E80287, the context need not be saved and restored. All that is necessary is that a selector for a new 94-byte data segment be written to A?MSR_SELECTOR, and the previous task's A?MSR selector be restored if that task is resumed. In this way, context switching can be made faster for the emulator than for the component. The contents of A?MSR are identical to the machine state stored with an FSAVE instruction, except that the stack registers are kept in a fixed order as though the top of stack pointer is always 0. Data in A?MSR may be referenced through the PUBLIC WORD PTR A?MSR_DATA. Knowing the format of segment A?MSR, the user may wish to manipulate the context of E80287 externally, without executing time-consuming floating point administrative instructions, but care must be taken to ensure that data is read from or written to the proper location.

Differences

There are a number of differences between the behavior of the 80287 and E80287 which are summarized as follows:

Floating point exceptions are raised by E80287, just as they are by the component. Whenever an unmasked floating point error is detected, the emulator executes an "INT 16" instruction and transfers control to the currently active floating point exception handler. Unlike the 80287, E80287 calls the interrupt handler immediately upon termination of the the emulated instruction. Also unlike the 80287, the

emulator leaves its own return status on the system stack when branching to the error handler. The IP, CS, and FLAGS for the return from the floating point instruction are underneath the three words stacked for the emulator's return. If an exception handler must locate the original floating point instruction return, it can simply discard the emulator status, since this simply points to an "IRET" instruction within the emulator.

FSETPM, FENI, and FDISI instructions are processed like FNOP by E80287.

Certain condition flags in the 80287 status word are set or reset by the 80287 processor in an undocumented way, but are left alone by the emulator except for those operations which explicitly redefine them.

Some transcendental instructions like FYL2X and F2XM1 may generate results for some operands which differ in the least significant bit for 80287 and E80287. These discrepancies are due to differences in the computational algorithms used by each floating point processor.

Memory protection violations which occur as the result of emulated instruction memory operations may point back to emulator code.

Related Publications

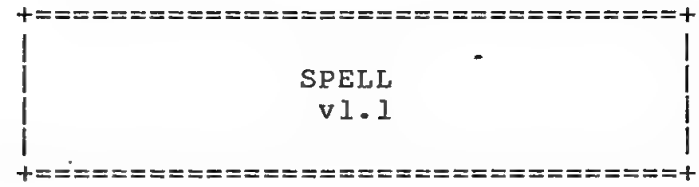
For complete information on the 80287 numeric data processor, refer to the following manuals:

- iAPX 286 Programmer's Reference Manual Numerics Supplement, 122164

Timing

This table lists typical execution times for selected floating point instructions on the specified numeric processors. E8087 is the 8087 emulator; 8087 is the hardware component; E87:87 is the ratio of E8087 to 8087 execution time; SIM286 is the iAPX-286 simulator; E80287 is the 80287 emulator; 80287 is the hardware component; E287:287 is the ratio of E80287 to 80287 execution time. All timings have been normalized for a 5 mHz clock and are given in microseconds.

INSTR\NDP	E8087	8087	E87:87	SIM286	E80287	80287	E287:287
FNOP	810	1.1	736:1	2140	300	2.5	120:1
FINIT	800	1.1	727:1	3580	260	2.1	124:1
FSAVE	1790	65.6	27:1	8510	400	48.3	8:1
FRSTOR	1450	68.3	21:1	4480	400	49.5	8:1
FLD1	1060	5.1	208:1	2400	540	8.4	64:1
FILD DW	2110	11.8	179:1	3990	1400	15.9	88:1
FILD DD	2840	14.5	196:1	4830	2320	16.2	143:1
FILD DQ	4840	16.9	286:1	6530	4170	19.4	215:1
FLD DD	1330	11.5	116:1	3110	510	12.9	40:1
FLD DQ	1500	13.3	113:1	3300	630	15.3	41:1
FLD DT	1330	14.5	92:1	3040	520	16.2	32:1
FBLD	11340	65.5	173:1	14490	8180	67.1	122:1
FLD ST	1600	4.7	340:1	2540	520	5.2	100:1
FADD ST,ST	1860	18.2	102:1	3170	940	22.5	42:1
FADD DQ	2330	20.1	116:1	4190	1190	22.7	52:1
FDIV DD	6440	18.0	358:1	8830	5320	20.0	266:1
FSTP DT	1350	12.2	111:1	3030	590	12.7	46:1
FBSTP	17970	104.8	171:1	21460	12970	106.1	122:1
FPATAN	20150	114.3	176:1	30180	17980	117.4	153:1
FXAM	1000	2.2	454:1	2330	460	3.3	139:1
FYL2X	28050	181.4	155:1	31780	18450	186.5	99:1
F2XM1	6850	121.4	56:1	8880	5190	127.3	41:1



SPELL is a simple 8086 program used to check for spelling errors in the given text files. The program tokenizes the input into words and looks up each word in its dictionaries. If the word is not found, SPELL will query the user to find out if it is a misspelling, a word to be ignored or a correctly spelled word which is not in the dictionary. All words are converted to upper case for comparison.

The main SPELL dictionary is maintained in a compressed form in the file SPELL.DC1 and must appear in the same directory as SPELL.86. A second word list file also used by the program for additional words is in the file SPELL.DC2 and must also be in the same directory. SPELL.DC2 is a simple ASCII file of words which are loaded into a secondary dictionary space for new words and user customization. This file may be edited by the user and is updated whenever a new correctly spelled word is encountered.

Seven additional compressed dictionaries can also be used by SPELL. These have the names SPELL.DC3, SPELL.DC4, ... SPELL.DC8, SPELL.DC9. As with DC1 and DC2, these must also be in the same directory. SPELL attempts to load these dictionaries in order, starting with DC3, then DC4, etc. through DC9. As soon as one of them is not found, SPELL stops the dictionary loading and does not attempt to load any of the subsequent dictionaries in the list.

All the dictionaries must fit into memory. If memory is exhausted while loading any dictionaries after SPELL.DC1, SPELL informs the user and begins checking the input text.

When an unknown word is found, SPELL will write out the line with the offending word and prompt the user with:

```
"word", R)ight, W)rong, (Any)=ignore?
```

where word is the string of letters under inspection.

Typing an "R" will result in the word being saved and later written to the SPELL.DC2 file. Typing a "W" will result in an error line being written to a file with the same name as the input file but with extension "ERR".

The error line is of the form:

```
LINE ###: "word"
```

where ### is the number of the input file line with the offending word and "word" is as above.

Typing any other character in response to the query will result in the word being ignored for the current execution.

The invocation syntax is:

```
SPELL option

option = spell-option | dump-dict-option .

spell-option = [ scribe-or-batch ] [ input-file-list ] .

scribe-or-batch = scribe-option [ "BATCH" ]
                | "BATCH" [ scribe-option ] .

scribe-option = "NOSCRIBE" | "SCRIBE" .

input-file-list = input_file [ input-file-list ] .

dump-dict-option = "(" output_file ")" .
```

The use of the SCRIBE switch indicates that all text between matching /'s is to be ignored and words immediately following a % are to be ignored. Also, the ^ literalizes the next character. The default is NOSCRIBE. (Refer to the description of SCRIPT in this manual.)

The BATCH switch allows SPELL to be used in a completely non-interactive manner. Instead of prompting for a Right, Wrong, or Ignore response, SPELL considers all unknown words to be Wrong, and writes them to the appropriate input_file.ERR file. This allows, for instance, the spelling check to be executed in a submit file, with the assumption that the user verifies the incorrectness of the words at a later time.

Notice that a list of input filenames may be processed. SPELL checks each file in the order given.

If no input text filenames are given, then the program reverts to an interactive mode in which SPELL prompts for each line from the console and checks words. This is useful for entering variations of words previously entered. An escape character (1BH) entered in response to the prompt terminates the input.

The dump-dictionary option is used to consolidate the secondary dictionary SPELL.DC2 into the main dictionary, in compressed form. SPELL writes to the specified output file the total contents of its dictionaries, as a list of ASCII format words, one per line, in alphabetical order. This file can then be input to the WSORT program (see description following) to create a new SPELL.DC1.

WSORT converts a list of words into the compressed dictionary format used by the SPELL program.

The input_file MUST consist of a list of words, ONE WORD PER LINE. A line must end with a line-feed character. The first letter of a word must be the first character on the line; there can be no leading blanks or tabs. A word consists of upper and lower case alphabetic characters, "A"..."Z" and "a"..."z", and the apostrophe character, "'". (The apostrophe is also known as the single quote, = 27H.) Lower case letters are converted to upper case.

Since WSORT alphabetically sorts the list, the words in the input_file may be in any order.

WSORT can process a MAXIMUM of 30000 words.
WSORT also limits the size of the compressed dictionary to 60000 characters.

The invocation syntax is:

```
WSORT infile  [ "TO" outfile ] [ "VM" "(" vm-level ")" ]
```

If "TO outfile" is not specified, then the output filename is the same as the input filename, but with extension "DC1".

The optional VM control allows the virtual memory size used by WSORT to be set to one of four values:

vm-level	virtual memory
Ø	128 KiloBytes
1	256 KB
2	512 KB
3	1 MegaByte

If vm-level is not in the range Ø through 3, then vm-level reverts to the default value of 1, or 256 KB virtual memory.

The size of virtual memory determines the maximum amount of space that WSORT uses for storing the input records. If the physical memory available is less than the virtual memory, then WSORT spills records to disk (:WORK:) if necessary. However, if spilling takes place, the sort is likely to be extremely slow.

USE RECOMMENDATIONS:

1. Only sort files of size less than the amount of available physical memory.
2. Set the virtual memory size to be as close to, but never less than, the size of the input file.

XREF produces an inter-module cross reference listing of symbols and numbers. The input to XREF is a set of list files, and the symbols are referenced to the line numbers in those files.

The allowable input files are PL/M listings and ASM(86) listings. Any other input is treated as plain data and is not processed.

The output format is:

```
<symbol>          <file name>   <line #>  <line #> .....
```

The symbols are listed in alphabetical (lexical) order.

The characters following some of the line number references in the cross reference have the following meanings:

```
S  ==>  symbol is set to a value at this line
*  ==>  symbol is defined as an entry at this line
Q  ==>  symbol appears within a quoted string at this line
D  ==>  symbol is defined at this line (EQU in asm)
```

XREF is invoked as:

```
XREF  input-list  [ controls ]
```

input-list is one or more filenames which are the list files to be cross-referenced. Filenames must be separated by commas. Wild card characters are allowed in the file specification.

*** CAUTION: use of wild card characters is only valid if the device referenced has an ISIS format file system.

controls allowed are:

PRINT (filename)

Specifies the file where the output from the XREF program is placed. The default filename is :F1:XREFL.LST. If the file specified cannot be

opened then the default filename will be used on the workfiles drive.

examples: PRINT(:F2:FN.XFR)
Output goes to :f2:fn.xfr.

PRINT(:F2:XY.LST) WORKFILES(:F3:)
If, for some reason, :f2:xy.lst
cannot be opened, :F3:XREFL.LST
is used for the print file.

EXCLUDING (symbol list)

Specifies a list of symbols which are not to be included in the cross reference. The elements of the symbol list may be separated by commas or spaces. The list is terminated by the closing parenthesis.

COMPRESS

Specifies that the output is to be compressed by removing the blank line which normally separates each symbol in the output.

TITLE ('title')

Specifies a title that is printed at the top of each page of the output. The title string is a sequence of up to 60 printable ASCII characters enclosed in quotes.

NONUMBERS

Specifies that numbers are NOT to be included in the cross reference output. If this control is not present then all numbers are cross-referenced along with all other symbols. Numbers appear in the output before symbols in lexical order.

WORKFILES (:Fn:)

Specifies the drive to contain the temporary files for XREF. The default is :F1:. The temporary files have names "SOAnnn.TMP", where nnn is "000" for the first temporary file needed, and increases by one for each subsequent temporary file created.

DOLLAR

Specifies that the dollar ($) is to be considered a significant character in symbols and numbers.

The default is to consider all dollar signs as transparent and not to include them in symbol names.

XREF8Ø is exactly like XREF, except that it processes ASM8Ø or ASM48 assembly language listings, instead of ASM86 listings.

PERF is a rather simplistic tool useful for determining the "hot spots" in a program's execution. It does this by partitioning memory into 1024 areas and sampling the program counter at a given rate to determine which area the program is executing in. At the end of the program or overlay, the count for each of the areas is displayed (only the non-zero ones).

Invocation:

PERF [output_file]

where output_file is optional and may be used to direct the output of the tool to a different file. :CO: is used by default.

PERF will then sign on and ask for a command line. Type in the normal invocation line (without RUN). The filename of the program being analyzed is assumed to have the ".86" extension. PERF will then ask for the low and high offsets for the measurement and for the rate of sampling. The offset values should be the offsets in the code segment for the area of interest. For example, to monitor the entire CS-relative addressing space, use 0 for the low offset and FFFF for the high offset. This implies that the entire 64K segment will be monitored with each area 64 bytes long. As smaller offsets are used, the areas will shrink accordingly. The RATE value is measured in milliseconds and represents the rate of the sampling. The default is one millisecond. For special cases, a tenth-millisecond rate can be generated by using FFFF for the rate.

If the program has overlays, PERF will, at every load overlay call, flush the results of the just completed sample, display the name of the overlay, and ask for new range and rate values. If it desired to not sample a particular overlay, set both the low and high range offsets to zero.

NOTE: This program will only monitor the starting CS segment, i.e. the CS value when the program is loaded. Therefore, it is not very useful for MEDIUM or LARGE case programs. It also will not work with LOCATEd programs since the initialization code is

built in a different segment.

The output is:

```
llllH - hhhhhH   nnnnH
```

for each of the 1024 areas (as long as they're non-zero) where llllH is the low end of the area,
hhhhH is the high end of the area and
nnnnH is the number of times the program counter was found to be in the area.

Also provided is the ELAPSED TIME, which is the total number of samples which occurred, and the SAMPLE TIME which is the sum of the counts for the areas.

This information is to be used with a link map to determine the locations that the program is spending its time.

*** WARNING ***

If the analyzed program has overlays, and if console input is set to transparent mode (with dq$special type 1 or 3) and not reset to line-edit mode (with dq$special type 2) before the next overlay call, then PERF may hang when prompting for the range information.

GRAFIT is a simple tool that creates a graph version of the output of the PERF tool, to allow quick visual inspection for 'hot' spots in the code.

GRAFIT is invoked as:

GRAFIT input-file

input-file MUST be the output_file generated by the PERF program.

The output-file produced by GRAFIT has the same name as the input-file with extension 'GRF'.

example.

RUN GRAFIT :F5:TSTFIL.PRF

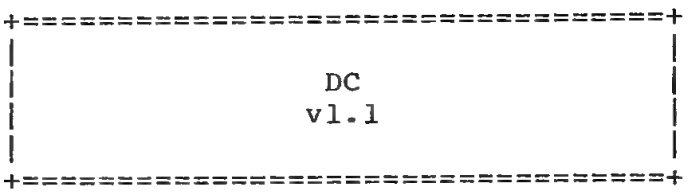
writes to :F5:TSTFIL.GRF the graph version of the Perf output file :f5:tstfil.prf.

The output of GRAFIT is intended to be printed on a line printer, as the width of the graph produced extends to a full 132 columns.

The following is an example GRAFIT output. Note that some of the lines actually extend farther right than is shown here.

```
LOWER RANGE: 1000
UPPER RANGE: 1400
RATE: ffff
ELAPSED TIME: 0002EA58H
  1000H - 1001H  0025H  ***
  1001H - 1002H  0009H
  1002H - 1003H  002BH  ****
  1003H - 1004H  001BH  **
  1005H - 1006H  0006H
  1007H - 1008H  000BH  *
  1008H - 1009H  0045H  ******
  100AH - 100BH  002FH  ****
  100CH - 100DH  001AH  **
  100FH - 1010H  0036H  *****
  1013H - 1014H  00B7H  ******************
```

```
  1015H - 1016H   0014H   **
  1019H - 101AH   002BH   ****
  101DH - 101EH   0025H   ***
  101FH - 1020H   000EH   *
  1022H - 1023H   0280H   ****************************************
  1024H - 1025H   0001H
  1025H - 1026H   0001H
  102CH - 102DH   0001H
  1034H - 1035H   0002H
  1038H - 1039H   0001H
  103CH - 103DH   0001H
  103EH - 103FH   0001H
  103FH - 1040H   0003H
  1041H - 1042H   0034H   *****
  1044H - 1045H   0010H   *
  1046H - 1047H   0002H
  104BH - 104CH   000AH   *
  104FH - 1050H   000AH   *
  1053H - 1054H   0007H
  1054H - 1055H   02B4H   ****************************************
  1056H - 1057H   00DFH   **********************
  1057H - 1058H   005AH   *********
  1059H - 105AH   0108H   ***************************
  105BH - 105CH   0003H
  105FH - 1060H   0008H
  1063H - 1064H   001FH   ***
  1064H - 1065H   00F3H   ************************
  1066H - 1067H   020AH   ****************************************
  1068H - 1069H   00C8H   ********************
  106CH - 106DH   0217H   ****************************************
  107FH - 1080H   02C2H   ****************************************
  1081H - 1082H   0105H   ***************************
  1083H - 1084H   0002H
  1085H - 1086H   0002H
  1089H - 108AH   001DH   **
  108CH - 108DH   0003H
  1095H - 1096H   0004H
  1099H - 109AH   001DH   **
  10ACH - 10ADH   000BH   *
  10AFH - 10B0H   0007H
  10B3H - 10B4H   0002H
  10B4H - 10B5H   000DH   *
  10B5H - 10B6H   0001H
  10BEH - 10BFH   0015H   **
  10C9H - 10CAH   00D5H   **********************
  10CBH - 10CCH   0127H   ******************************
  10CDH - 10CEH   0265H   ****************************************
  10CEH - 10CFH   0004H
  10D2H - 10D3H   01DAH   ****************************************
  10D6H - 10D7H   0395H   ****************************************
  10E1H - 10E2H   0003H
SAMPLE TIME: 00002063H
```



DC is a simple floating-point Desk Calculator. DC supports a simple variable set, nested assignments, the four arithmetic operations addition, subtraction, multiplication, and division, plus exponentiation and the functions square root, exp, natural log, sine, cosine, and arctangent, and user defineable number base and width of decimal field.

Input to DC is from the console keyboard; output is to the console screen. DC reads a line which contains an expression to be evaluated and writes the result on the next line, continuing in this manner until the eof line is read.

INPUT SYNTAX

```
input = statement-list .

statement-list = statement [ statement-list. ] .

statement = end-of-input-statement
          | expression end-of-line .

end-of-input-statement = "#" end-of-line .

end-of-line = [ ";" comment ] EOLN .

EOLN = (* end-of-line character(s). *)

expression = [ add-op ] term [ add-op term ]...

add-op = "+"
       | "-" .

term = power-factor [ mul-op power-factor ]...

mul-op = "*"
       | "/" .

power-factor = factor [ power-op factor ]...

power-op = "^" .
```

```
factor = variable
       | number
       | "(" expression ")"
       | builtin "(" expression ")" .

variable = id
         | assignment .

id = id-char [ id-char ]...

id-char = (* a single upper or lower case letter *)
        | the character "@"
        | the character "?" .

assignment = id "=" expression
           | id ":=" expression .

number = fixed
       | string
       | float .

fixed = digits [ "." [ digits ] ]
      | [ digits ] "." digits .

digits = leading-digit [ digit ]...

leading-digit = "0" | "1" | "2" | "3" | "4"
              | "5" | "6" | "7" | "8" | "9" .

digit = (* single character valid for current number base *)

string = "'" [ sequence of characters ] "'" .

float = fixed(base 10) "E" [ add-op ] fixed(base 10) .

builtin = "SQRT"
        | "EXP"
        | "LN"
        | "SIN"
        | "COS"
        | "ARCTAN" .
```

IDENTIFIERS

An identifier is any string of letters, where a letter is an upper or lower case alphabetic character "A" through "Z", or the at sign character "@", or the question mark "?". The lower case characters are equivalent to the upper case characters. If the identifier is not equal to one of the builtin functions, then only the first character of the

identifier is significant. Thus, DC supports a set of 28 identifiers: "?", "@", "A"..."Z".

SPECIAL IDENTIFIERS

The identifier "@" represents the current numeric base used for input and output. The base may range from 2 through 36. In order to change base, "@" must be assigned the value of the base desired. The assignment to "@" is always made; however the base is changed only if the rounded value of "@" is in the correct range.

The identifier "?" represents both the format of the numeric result printed as output, and the number of fractional digits displayed (see the section OUTPUT below). The valid range of integers for controlling output is -18 through +18. The assignment to "?" is always made; however the output display format is changed only if the rounded value of "?" is in the correct range.

Both "@" and "?" may appear in any context where an identifier is allowed.

INITIAL VALUES OF VARIABLES

The 26 alphabetic identifiers A through Z have initial values of zero.
"@" is initialized to ten (decimal).
"?" is initialized to +2.

FLOATING POINT INPUT

Numbers input in floating-point representation are only valid if the base is ten. If the base is not ten and less than fifteen, an error is reported, since the digit "E" is not valid. If the base is fifteen or greater, the "E" is interpreted as a valid digit terminating the significand, rather than the beginning of the exponent part. For example, the input 1.1E+4 in base 16 is interpreted as (1.1E)+(4) with the result 5.1E (hex), instead of the desired result 11000.

NON-DECIMAL INPUT

It is noted that input numbers must start with a decimal digit, to avoid ambiguity with the variable ids.

ASSIGNMENTS

Any number of assignments may appear anywhere in an expression. The assignment operators "=" and ":=" are equivalent.

INPUT LINES

The length of an input line is limited to 76 characters, including comments. A comment begins with an unquoted semicolon and terminates with the end of line.

OUTPUT

The numeric result of the expression is printed as output on the line following the input. Intermediate results for embedded assignments are not displayed; the variables assigned must be entered individually on subsequent lines for displaying their values.

If the value of the result is exactly zero, the single digit "Ø" is displayed.

If "?" is negative, then all output is displayed in floating-point representation. Floating-point output is always base ten, no matter what the "@" base is. The absolute value of "?" specifies the number of decimal digits displayed.

If "?" is non-negative, then the value of the result determines the format of the output. If the absolute value of the result is greater than or equal to .000000000046566128 and less than or equal to 2147483647 then the result is displayed in fixed-point representation. If the base is non-decimal, the base is displayed, in parentheses, following the result. If the absolute value of the result is not in the above range, the result is displayed in floating-point decimal representation. The absolute value of "?" specifies the number of fractional digits displayed, except in the case where the number of fractional digits plus the number of integral digits is greater than nineteen. In this case, the fractional part is truncated such that the total number of significant digits is nineteen.

If the value of the result is positive infinity, a string of "+" is displayed; if the result value is negative infinity, a string of "-" is displayed; if the result value is a NAN, a string of "." is displayed.

SAMPLE SESSION

```
run :f1:dc
Desk Calculator (DC), V1.1
Copyright 1983 Intel Corp.
    0           ; NOTE THAT INPUT IS ON THE INDENTED LINES
 0
    0           ; AND OUTPUT IS ON THE NON-INDENTED LINES
 0
    ?=18                          ;set maximum decimal places
 18.000000000000000000
    a=1.1111111111111111
 1.1111111111111110
    b=a*a*a*a*a*a*a*a*a*a
 2.8679719907924412 86
    c=1/b
 0.3486784401000000 03
    d=2147483647                  ;maximum output integer
 2147483647.000000000
    d+1                           ;should print as floating-point
 2.1474836480000000 00E+0009
    e=1/d                         ;minimum output integer
 0.000000000465661288
    1/(d+1)                       ;floating-point output
 4.6566128730773925 80E-0010
    z=10*9*8*7*6*(y:=5*4)*3*2     ;shows value of z only
 3628800.000000000000
    y                             ;y should equal 20
 20.000000000000000000
    @=32                          ;set base to 32
 10.000000000000000000  (32)
    d                             ;base 32 appears in parens
 1VVVVVV.000000000  (32)
    e
 0.000000G00000800000  (32)
    @=0a                          ;set base back to 10
 10.000000000000000000
    ?=1                           ;one decimal place
 1.0
    c
 0.3
    ?=9                           ;nine decimal places
 9.000000000
    c
 0.348678440
    ?=18
 18.000000000000000000
    q=1.1e+4                      ;floating-point input
 11000.000000000000000
    r=1.1e-4
 0.000110000000000000
```

```
    @=16
 10.00000000000000000  (16)
    s=1.1e+4                    ;try it in hex
 5.1E0000000000000000  (16)
    t=1.1e-4
-2.E2000000000000000  (16)
    @=0a
 10.00000000000000000
    s
 5.117187500000000000
    t
-2.882812500000000000
    ?=-12                       ;subsequent output is
-1.200000000000E+0001           ;floating-point for all values
    a
 1.111111111111E+0000
    b
 2.867971990792E+0000
    c
 3.486784401000E-0001
    d
 2.147483647000E+0009
    e
 4.656612875246E-0010
    y
 2.000000000000E+0001
    z
 3.628800000000E+0006
    ?=18                        ;allow fixed-point output
 18.000000000000000000
    pi=arctan(1)*4              ;approximate pi
 3.141592653589793238
    cos(p*2)                    ;id "pi" is variable "p"
 1.000000000000000000
    sin(p*2)
 0
    cosine*1                    ;should be c=0.34867...
 0.348678440100000003
    pi*1e5                      ;only 19 significant digits
 314159.2653589793239
    2^2^2^2^2                   ;powers
 65536.000000000000000
    10^10^10^10^10              ;positive infinity
 ++++++++++++++++++++++++++
    -10^100^100                 ;negative infinity
 --------------------------
    ln(-9)                      ;NAN
 ..........................
    #                           ;exit
Exit
```

DC87 is exactly like DC except that it uses the 8087 hardware instead of the 8087 emulator. If an 8087 is present in the system, then DC87 can be used for faster results.

The program FUNC reads in a text file and creates a FUNCtion keymap from it for the Series IIIE. The invocation should look like this:

[RUN] FUNC [<file-name> | RESET]

If a file name is specified, that file will be read in as the keymap file. If the RESET command is used, the FUNCtion keys will be reset to the values provided on system reset. If no file name or command is provided, the file FUNC.DEF will be read in as the keymap file. FUNC.DEF must reside on the same disk as FUNC.86.

Here is an example of a function definition file: (do not use dashes)

```
A run aedit
C r calc

E exit

P pscope ws(32768) db(16)

Q QWERTY
q qual
z zing
Z ZOO
```

The first character of a line must be the key to be defined. A blank space follows. The next N (< 255) characters is the text that will be inserted when FUNC+<key> is typed. All characters up to the CR,LF will be included. For example, in the 'A' definition above, there is a blank space after 'aedit' so that a file name can be entered.

If the key to be defined is an upper case letter, the definition will be inserted for both upper and lower case versions of this function key. Thus, in the 'Q' definition above, both func-'Q' and func-'q' are defined as 'QWERTY'; the following 'q' definition redefines func-'q' as 'qual'.

Func-'Q' retains the 'QWERTY' definition. The 'z' definition defines only func-'z' as 'zing', func-'Z' retains its previous definition, if any. The following 'Z' definition both defines func-'Z' and redefines func-'z' as 'ZOO'.

To allow a carriage return to be inserted as part of the function text, place a second CR,LF pair after the definition as in 'C', 'E', and 'P' above. Multiple line definitions are not allowed.

The total number of characters in all definitions may not exceed 800.

Function definitions are not additive so calling FUNC twice in a row may not have the desired effect, however, all default (IOC provided) function keys that are not redefined will remain in effect.

```
+==================================+
|                                  |
|              ESORT               |
|               v1.1               |
|                                  |
+==================================+
```

ESORT sorts arbitrarily large files, using a partial sort and merge technique. It can sort by multi-keys (fields of columns), and allows record lengths up to 5118 characters. It also allows continuation lines within single records.

A line in the input file is a string of characters up to and including a CRLF, a carriage-return,line-feed pair.

ESORT is invoked as:

```
ESORT <input-file> [ TO <output-file> ] [ <up-down> ]
                   [ CC ( '<continuation-character>' ) ]
[ F[IELD] ( <range1> [ , <range2> [ ,..., <range10> ] ] ) ]
```

Where

<input-file> is the file to be sorted

<output-file> is the sorted file

<up-down>
is either UP or DOWN, and means the output file should be ordered in ascending or descending order

<continuation-character>
is a printable character (ASCII code 20H to 7EH) that will cause each line that begins with this character to be sorted with the last line previous that does not begin with this character. For example if the continuation character is an asterisk then:

```
line1
*line2
*line3
...line 4
*line 5
```

will be treated as two sortable units (or records), the first beginning with line1 and the second beginning with ...line 4. The CRLF and the continuation character are counted as part of the record.

<range>
is Ni:Nj specifying a sort field starting at column Ni and going to column Nj. (Ni must be less than or equal to Nj).

Defaults:

output file	<input-file>.SRT
up-down	UP (i.e. ascending order)
continuation character	none
field	1. If nothing is specified then there is one sort field: column 1 to the end of the record. 2. If the <range> is a single number N then that sort field starts at column N and goes to the end of the record.

Note that to use a single column as the sort field, the column number must be entered twice. For example, FIELD(...,3:3,...) will use only column three for this sort field. The precedence of the sort fields is the same as their order of entry.

The first column of a record is numbered as column 1.

An overlapping of the fields is permitted.

In case that a record does not contain part of a key or the whole key, it is declared to be less then a compared record that contains that key (it is treated as if it is expanded by nulls).

Error Messages

SYNTAX ERROR
an illegal control or illegal delimiter

TOO MANY ARGUMENTS
more than 10 fields were entered

WRONG RANGE
the last range read is illegal, the last column of the field is less than its first column, or the column is zero or larger than 5118

ILLEGAL CONTINUATION CHARACTER
    the continuation character is null or is longer than one character

RECORD TOO LONG
    the program contains a record larger than 5118 bytes

HSORT is a simple 8086 program used to sort the lines of a file into ASCII lexical order. HSORT uses the "heap sort" method.

A "line" in the file to be sorted is defined as a string of characters up to and including a line feed character. The maximum line length is 256 characters. If a line is longer than 256 characters, the first 256, or multiple of 256, characters are ignored, and the remaining characters on the line are sorted.

All characters on a line are significant, including unprintable ASCII characters.

HSORT does NOT allow multiple field sorting. Each line is considered to be a single field, starting with the first character on the line.

HSORT allows a maximum of about 30000 lines.

HSORT is invoked as:

```
HSORT infile [ "TO" outfile ] [ "VM" "(" vm-level ")" ]
```

If "TO outfile" is not specified, the output is written to a file with the same name as infile, but with extension "SRT".

The optional VM control allows the virtual memory size used by HSORT to be set to one of four values:

vm-level	virtual memory
0	128 KiloBytes
1	256 KB
2	512 KB
3	1 MegaByte

If vm-level is not in the range 0 through 3, then vm-level reverts to the default value of 1, or 256 KB virtual memory.

The size of virtual memory determines the maximum amount of space that HSORT uses for storing the input records. If the physical memory available is less than the virtual memory, then HSORT spills records to disk (:WORK:) if necessary.

However, if spilling takes place, the sort is likely to be extremely slow.

USE RECOMMENDATIONS:

1. Only sort files of size less than the amount of available physical memory.
2. Set the virtual memory size to be as close to, but never less than, the size of the input file.

If it is desired to sort files larger than the size of available memory, or to do multiple field sorts, then use ESORT, which is described elsewhere in this manual.

I. INTRODUCTION

A. PURPOSE

PASSIF was developed to automate the software testing process.

Such testing involves (among other things) searching files for specific strings, or the lack thereof, checking whether two files (usually object modules) are identical, and checking whether specified files exist.

In the past, performing and checking this style of test required the use of many different tools and utilities, and the examination and processing of huge disk files.

PASSIF obviates this by directly performing the test-checking functions needed, and reporting the results in a processed and compact form.

B. PHILOSOPHY

It is intended that PASSIF be able to adequately process the great majority of test cases encountered in a normal test suite. It is not intended to handle all conceivable requirements of this style of software testing.

It is assumed that PASSIF will typically be used to quality assure a large test base, run as a night job. PASSIF, therefore, is not a performance crucial program but its speed should be adequate for occasional interactive use.

Under exceptional conditions, PASSIF will attempt to continue processing, if there is reasonable method of doing so.

The design of PASSIF emphasizes simplicity and maintainability.

C. EXECUTION ENVIRONMENT

PASSIF runs in an 8086-based UDI-compatible environment, specifically Series-III and Series-IV.

II. SPECIFICATION

A. GENERAL METHOD OF OPERATION

1. Overview

PASSIF ascertains what assertion to check by reading its command tail. PASSIF then performs whatever functions are necessary to check the verity of the assertion.

The results of the assertion checking are then reported in a disk file called ":f1:report.log". This file has a "banner" at the top with three decimal ascii numbers, representing tests executed, tests passed, and tests failed, respectively. The numbers are updated by PASSIF during each of its invocations.

The report file is always opened in update mode. If the assertion(s) are true, then PASSIF merely increments the "TESTS PASSED" number and the "TESTS EXECUTED" number.

If the assertion is false, then the "TESTS EXECUTED" and "TESTS FAILED" numbers are incremented, and a "failed assertion entry" for the test which failed, is appended to the end of the report file. A sample of a typical report file, and a detailed description of what type of information appears in a "failed assertion entry" appears below.

Each time it is invoked, PASSIF dumps some test status information to the cold boot console. As soon as it comes up, it prints a carriage-return (but no line-feed). When PASSIF is finished processing, it prints out the new banner it has generated, but leaves off the carriage-return line-feed at the end.

Thus, the user sees a periodically updated display of test status information on his console. He knows, in real time, when PASSIF is running (cursor at the beginning of the line), and when the rest of the test suite is running (cursor at the end of the line). The display of this information only costs one line of screen, which is painted over and over again, so the user won't lose status information which any other of his programs may have dumped to the console.

EXAMPLE OF A TYPICAL REPORT FILE

```
================================================================

13517 TESTS EXECUTED   13499 TESTS PASSED   00018 TESTS FAILED

----------------------------------------------------------------

test 00533 failed
    PASSIF.86 STRING_FOUND(ERROR 28)IN FILE(:F1:DC.MP1)

----------------------------------------------------------------

test 00534 failed
    PASSIF.86 STRING_FOUND(ERROR 29)IN FILE(:F1:DC.MP1)

----------------------------------------------------------------

test 00714 failed
    Passif invoked by:
    PASSIF.86 FILE_EXISTS(FOOOBAR.CS)#
UDI error 010CH encountered while performing assertion checking

----------------------------------------------------------------

test 00998 failed
    Passif invoked by:
    PASSIF.86 STRING_FOUND(ERROR 28)IN FILE(:F1:LOC86.MP2)#
UDI error 0021H encountered while performing assertion checking

----------------------------------------------------------------
================================================================
```

2. Test Failure Reporting

When a test fails, PASSIF appends an entry to the report file specifying:

a. the name of the console input file
b. the ordinal number of the test
c. the assertion which caused the failure

3. Test Result Report File

The results of the tests performed by PASSIF will be reported in file ":F1:REPORT.LOG".

The use of a "hardwired" output file contributes to the parallelism and understandability of testing procedures between users of PASSIF.

Other methods of specifying the report file are less desirable. If the report file is specified in a "macrofile" which is read by PASSIF upon each of its invocations, a substantial performance penalty is incurred, especially with a hard disk system.

If the report file is specified in PASSIF's command tail with a control or switch, then the command tail becomes cluttered, and loses its "readable-English" flavor.

The user may change the report file specification in PASSIF, if absolutely necessary, by editing the load module, since the report file is specified therein by the string "REPORT_FILE=:F1:REPORT.LOG". The difficulty of changing the report file specification is deliberate.

4. Exceptional Conditions

a. Unreadable Report File

Since PASSIF must read the report file banner to "orient" itself (ascertain the current ordinal test number), an unreadable banner constitutes an exceptional condition.

In such cases PASSIF will "re-initialize" the report file by inserting a new banner at the beginning of the report file. Whatever was already in the report file will, therefore, not be destroyed.

The values of "TESTS PASSED", "TESTS FAILED", and "TOTAL TESTS EXECUTED" are all "initialized" to zero in the new banner.

b. Syntax Error in Command Tail

If there is a syntax error in PASSIF's command tail, then a "test failed" entry is made in the report file. The string "syntax error in command tail:" appears in the report file, followed by the offending command tail, up to the point where the error was detected, followed by a hash-mark. This is the standard command-tail error reporting mechanism used by most Intel products.

B. EXAMPLES OF LEGAL COMMANDS WITH EXPLANATIONS

1. File Exists

Example:

```
PASSIF FILE_EXISTS(:F1:T23.MP2)
```

Action:

PASSIF checks to see whether file ":f1:t23.mp2" exists. If the file does not exist, or the file exists, but has zero length, then a "failed assertion" response is triggered; else, the test passes. A file of zero length is considered to be a failure because the existence of such a file is almost always an indication of an exceptional condition. However, if the file is a spool queue file, e.g. :SP:T45.LST, the length of the file is not considered.

2. File Absent

Example:

```
PASSIF FILE_ABSENT(:FØ:ERRORS)
```

Action:

PASSIF checks that file ":fØ:errors" does not exist. Detection of a file of zero length, or greater, triggers a "failed assertion" response.

3. Files Match

Example:

```
PASSIF FILES_MATCH(:F1:T31A.OBJ,:F6:T31A.OUT)
```

PASSIF checks whether the specified files are identical.

4. String Found

Example:

```
PASSIF STRING_FOUND("*** ERROR #217") IN FILE (:F1:P11TØ1.LST)

PASSIF STRING_FOUND &
("this is quite a long string which contains a """) &
IN FILE (:F1:P11TØ1.LST)
```

PASSIF checks whether the specified string appears in the specified file. Notice that ampersand continuation is allowed between tokens, as per usual.

5. String Absent

Example:

```
PASSIF STRING_ABSENT("ERROR") IN FILE(:F1:T22A.MP1)
```

PASSIF checks whether the specified string is absent from the specified file.

C. ADDITIONAL SEMANTICS FOR COMMAND TAIL SPECIFICATION

1. Keyword Abbreviations

The keywords may be abbreviated as follows:

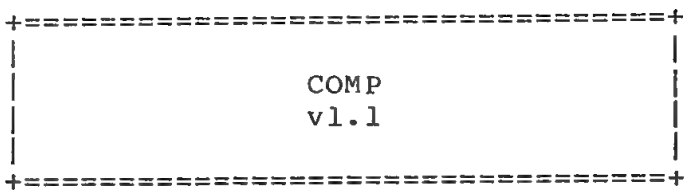
FILE_EXISTS	==>	FE
FILE_ABSENT	==>	FA
FILES_MATCH	==>	FM
STRING_FOUND	==>	SF
STRING_ABSENT	==>	SA

2. Multiple Commands

Multiple commands are not allowed in a single invocation of PASSIF. The only multiple entities (lists) allowed in a single command are the two filenames required in the "files_match" command.

3. Null Command Tail

If the command tail to PASSIF consists only of blanks and tabs then PASSIF signs on and exits.



1. SUMMARY

COMP compares two files (text or object) and displays the differences between them.

COMP may be used to compare files which contain substantial differences. It is fast enough that finding a single byte mismatch is feasible.

2. SAMPLE INVOCATION

```
COMP  :f1:good.fil to :f1:bad.fil  &
                print(:f1:output.cmp)  sync(3)
```

In this example, the two text files are compared, and the portions which miscompare are displayed in file ":f1:output.cmp". "Sync(3)" means that 3 lines must match before COMP deems that the files have been resynchronized.

3. ACTIONS PERFORMED

COMP prints out the mismatching blocks of lines. For text files, the text is printed out following the line number. For object files, the hex representation of the record is printed out following the record number.

4. OUTPUT FORMAT

Each block of mismatched lines or object records is printed with the file name as the first line. Each succeeding line begins with the line number or the record number of the mismatched item. This is followed by the text of the line or the hex representation of the object record (sixteen bytes per line).

5. INPUT FORMAT

5.1 Text

Text files are treated as having the line feed character (0AH) as the last character of each line. If the length of the longest line exceeds the buffer size, an error occurs (see the SIZE option).

5.2 Object

Object files are a sequence of hex bytes. They are treated as a sequence of records, each having the form of a one byte type field, followed by a one word length field (low byte then high byte), followed by the remainder of the record. This remainder consists of the number of bytes in the length field. The size of the largest record (3 + the maximum length field) must not exceed the buffer size (see the SIZE option). If this happens, it is an error.

6. EXAMPLE

Given the "sample" command line specified above, and given that the text files to be compared contain:

good.fil

```
+-----------------------+
|      good line 1      |
|      good line 2      |
|      good line 3      |
|      good line 4      |
|      good line 5      |
|      good line 6      |
|      good line 7      |
+-----------------------+
```

bad.fil

```
+-----------------------+
|      bad line 1       |
|      good line 2      |
|      good line 3      |
|      good line 4      |
|      good bad line 5  |
|      good line 6      |
|      good line 7      |
|      extra line       |
+-----------------------+
```

then the following output will be placed in file ":f1:output.cmp":

```
============================================================
FILE # 1: :F1:GOOD.FIL
    1: good line 1
    2: good line 2
============================================================
FILE # 2: :F1:BAD.FIL
    1: bad line 1
    2: good line 2
============================================================
============================================================
FILE # 1: :F1:GOOD.FIL
    5: good line 5
    6: good line 6
    7: good line 7
============================================================
FILE # 2: :F1:BAD.FIL
    5: good bad line 5
    6: good line 6
    7: good line 7
    8: extra line
============================================================

FILES DIFFER
```

7. INVOCATION SYNTAX

```
COMP  <filename>  TO  <filename>
      [ PRINT ( <filename> ) ] [ SYNC ( <number> ) ]
      [ SIZE ( <number> ) ] [ OBJECT ] [ ICR ]
```

<number> is a decimal number in the range allowed by the control.

7.1 PRINT Control

If the print control is present in the command line, then output is directed to the specified file. The default is to send the output to :CO: . The control abbreviation PR is also allowed.

7.2 Synchronization (SYNC) Control

The sync control specifies the number of lines (or records) that must match before COMP deems that the files have synchronized. The default synchronization value is 3. The value must be in the range of 1 to 255. Any larger value will be truncated to this range. A zero value will be ignored, causing the default to be used.

7.3 Buffer Size (SIZE) Control

This sets the size of the program buffers for the two input files and is useful if comparing files with unusually long lines (or object records) or an unusually long stretch of differences. Each line (or record) must fit in the buffer. If differences are encountered between the two files, those differences plus the synchronization lines (or records) must fit in the buffers. Failure of either of these will cause COMP to abort with an error message.

The default buffer size is 8192 bytes. The parameter for this control must be within the range of 1024 to 65535. Any larger values will be truncated to a word value. Any smaller values will be set to 1024. Note that performance will be better if the value selected is a multiple of 512.

COMP will attempt to allocate, using DQ$ALLOCATE, two buffers, each of the size indicated. Insufficient memory in the system will cause DQ$ALLOCATE to indicate an error.

7.4 OBJECT Control

This control indicates that the two files to be compared are object files as described in section 5.2 . The default is to compare the files as text files as described in section 5.1 . The abbreviation OJ is allowed.

7.5 Ignore Comment Records (ICR) CONTROL

This control is used only when the OBJECT control is in affect. It instructs COMP to ignore the contents of comment records when comparing two object files. This allows a clean compare to occur when comparing files whose only difference is the version of the compiler that generated the object (compiler name and version number are typically stored in a comment record near the beginning of the object file). Note that if a comment record is present in one file and absent in the other, a difference will be flagged. It is the contents that is ignored, not the presence or absence of the record itself.

7.6 Help (?)

The question mark control '?' causes COMP to print out a short description of invocation parameters.

8. ERROR MESSAGES

```
INSUFFICIENT ROOM TO SYNCHRONIZE
   TRY A LARGER BUFFER SIZE (SIZE OPTION)
```

This indicates that the number of bytes needed to contain a string of differences plus the synchronization lines (records) was greater than the buffer size. Use the SIZE option to create larger buffers.

```
INSUFFICIENT ROOM FOR LARGEST RECORD
   TRY A LARGER BUFFER SIZE (SIZE OPTION)
```

This indicates that the number of bytes needed to contain the longest line (or largest record) was greater than the buffer size. Use the SIZE option to create larger buffers.

```
BAD OPTION PARAMETER: #
   DECIMAL NUMBER EXPECTED
```

The SYNC and SIZE options require a decimal value as a parameter. The option scanner encounterred a non-decimal character in the parameter.

intel

REQUEST FOR READER'S COMMENTS

Intel's Technical Publications Departments attempt to provide publications that meet the needs of all Intel product users. This form lets you participate directly in the publication process. Your comments will help us correct and improve our publications. Please take a few minutes to respond.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this publication. If you have any comments on the product that this publication describes, please contact your Intel representative. If you wish to order publications, contact the Intel Literature Department (see page ii of this manual).

1. Please describe any errors you found in this publication (include page number).

2. Does the publication cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of publication for your needs? Is it at the right level? What other types of publications are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this publication on a scale of 1 to 5 (5 being the best rating).

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

CITY _____ STATE _____ ZIP CODE _____

(COUNTRY)

Please check here if you require a written reply. ☐

WE'D LIKE YOUR COMMENTS ...

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

NO POSTAGE
NECESSARY
IF MAILED
IN U.S.A.

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1040 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
Attn: Technical Publications M/S 6-2000
3065 Bowers Avenue
Santa Clara, CA 95051

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) 987-8080

Printed in U.S.A.